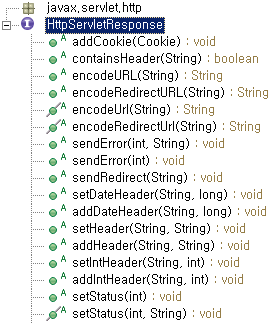
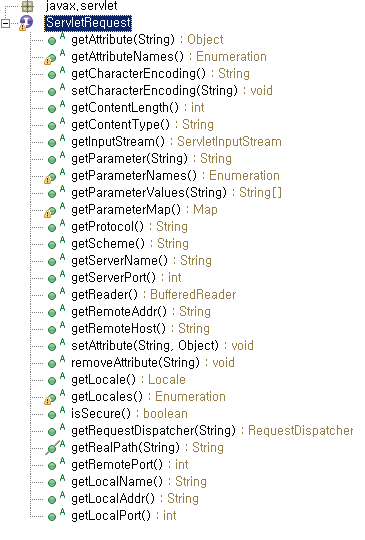
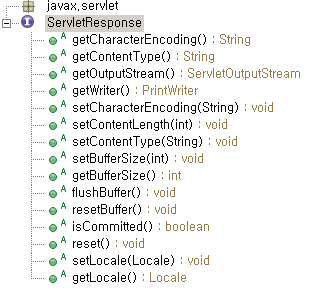
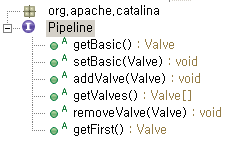
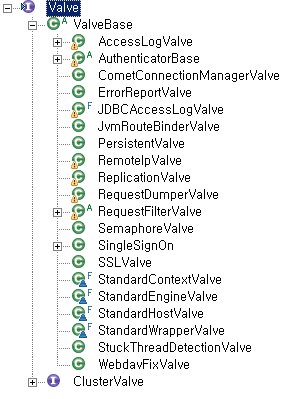
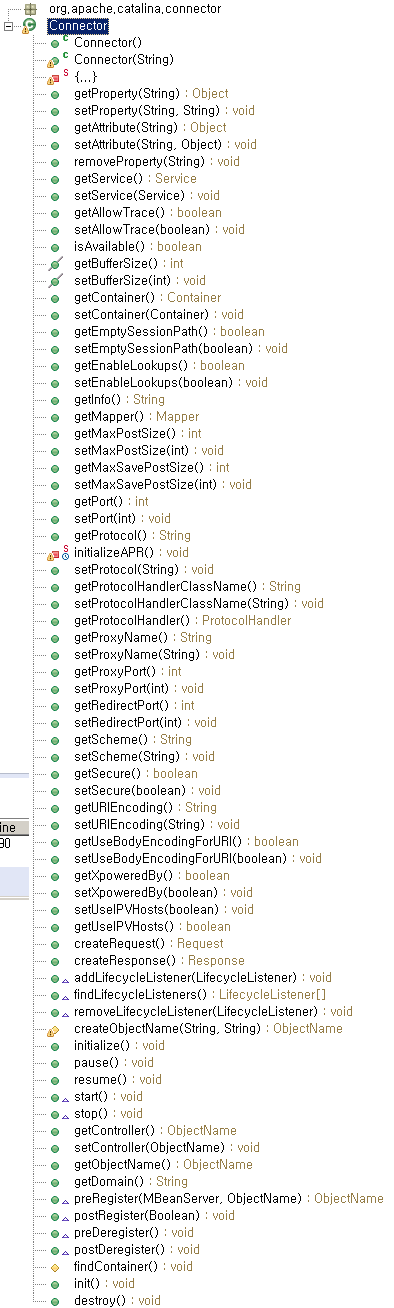
61ㅠㅠㅠㅠㅠ

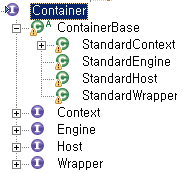


Container





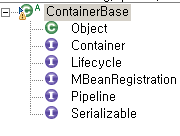
org.apache.catalina.Container (Engine, Host , Context， Wrapper)

<-중요

,





<-중요

서블릿 요청을 처리하고 웹 클라이언트에게 보낼 응답 객체를 만들어 주는 컴포넌트이다.

**• 엔진(Engine ) 은 카탈리나 서블릿 엔진 전체를 나타낸다**

**• 호스트 (Host) 는 다양한 컨텍스트에 따른 가상 호스트를 나타낸다.**

**• 컨텍스트 (Context) 는 웹 애플리케이션을 나타내며 둘 이상의 래퍼를 가진다 .**

**• 래퍼 (Wrapper) 는 각각의 서블릿을 나타낸다.**

코요테(Connector) org.apache.catalina.connector.Connetor

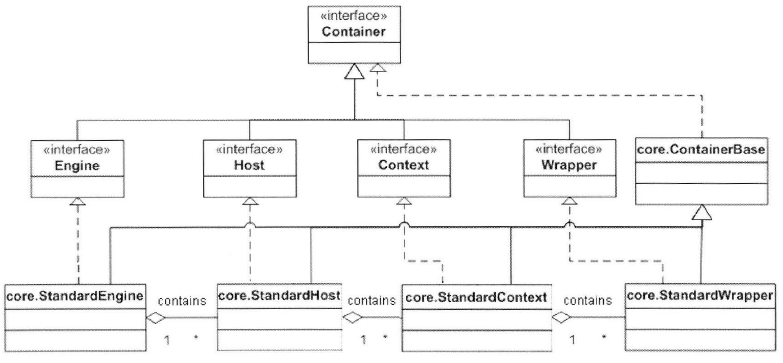
1. org.apache.catalina.Connector 인터페이스 구현

2. org.apache.catalina.Request 인터페이스를 구현하는 요청 객체 생성

3. org apache.catalina.Response 인터페이스를 구현하는 응답 객체 생성

요청이들어오면

CATALINA



**public** Loader getLoader() {

**if** (loader != **null**)

**return** (loader);

**if** (parent != **null**)

**return** (parent.getLoader());

**return** (**null**);

}

Connector

파이프라인은 필터 체인과 같고， 밸브는 펼터와 같다. 필터와 마찬가지로 밸브는 전달받은 요청/응답 객체를 조작할 수 있다. 하나의 밸브가 처리 작업을 끝내면 파이프라인에서의 다음 밸브를 호출 하게 된다. 기본 밸브는 항상 마지막에 호출된다

파이프라인org.apache.catalina.Pipeline





pipeline.getFirst().invoke(request, response);

서블릿 클래스의 로드 service 메소드 호출， 세션 관리， 에러 메시지의 로깅 등 다양한 일을 한다

Container





요청

클라

이언트

-44



StandardPipeline

Valve first;

**public** Valve getFirst() {

**if** (first != **null**) {

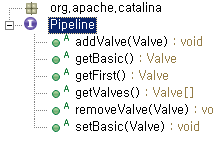
**return** first;

} **else** {

**return** basic;

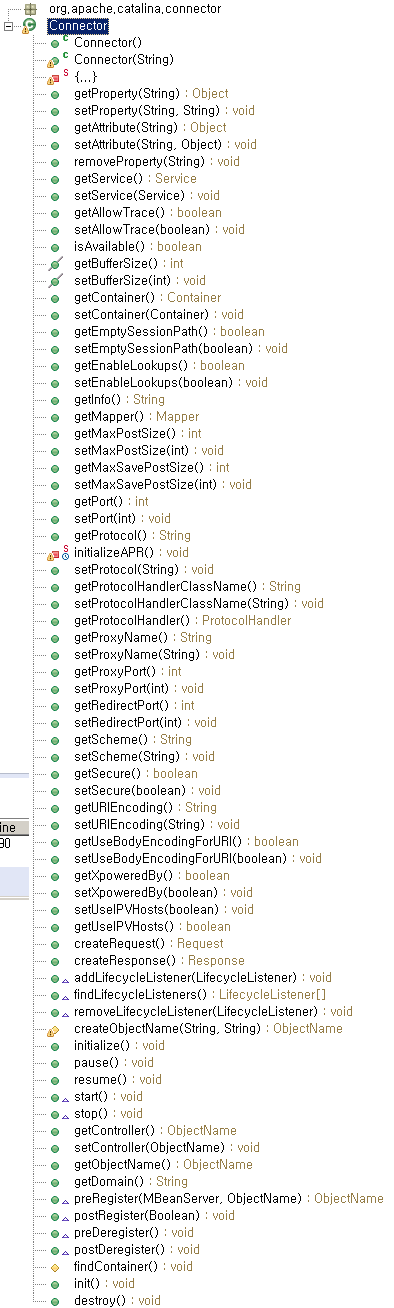
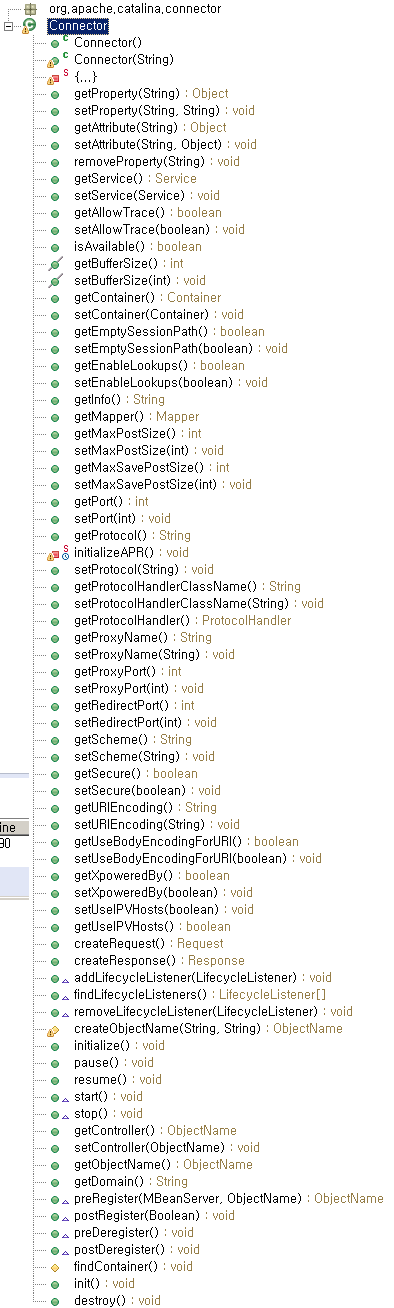
}

}



1 1

1 1



**Context 컨텍스트는** 웹 애플리케이션을 나타내는 컨테이너이다 컨테이너는 하위 컨테이너로서 보통 둘 이상의 래퍼를 갖고 있다.

Context 인터페이스의 중요한 메소드로는 **addWrapper와 createWrapper (Wrapper는 서블릿을 말한다)**

대부분의 웹 애플리케이션은 다수의 서블릿을 갖는다. 이 같은 경우 래퍼와는 다른 종류의 컨테이너가 필요한데， 그것이 바로 컨텍스트이다

\*애플리케이션에서는 2개의 서블릿 클래스를 각각 래펑하고 있는 2개의 래퍼가 존재하며 ， 이2개의 래퍼는 하나의 컨텍스트에 포함된다 둘 이상의 래퍼를 가지려면 ~~맵퍼 (mapper) 라는 것이 필요한데 ， 맵퍼는 컨테이너 (여기서는 컨텍스트)가 특정한 요청을 처리할 하위 컨테이너를 선택하는 것을도외주는 컴포넌트이다.~~



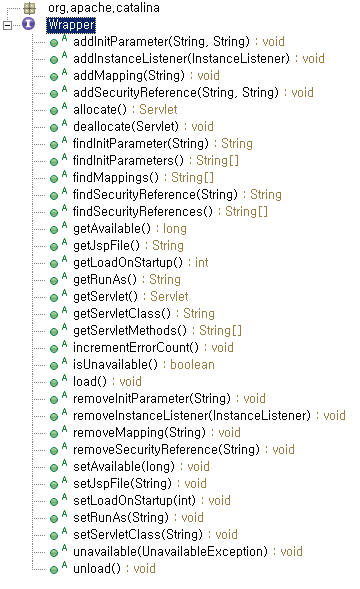
**Context는 Wrapper만 자식으로 가질수 있다. setServletMapping으로 Wrapper별로 서블릿 맵핑을 시켜줄수있다.**

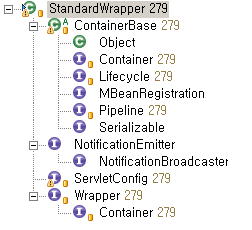
**( addServletMapping 메소드는 URL 패턴/래퍼의 이름을 추가한다.)**

Wrapper 인터페이스는 하나의 래퍼를 나타낸다. 래퍼는 개별적인 하나의서블릿을 대변하는 컨테이너이다 allocate와 load매서드가 중요하다.

**load()에서-> StandardWrapper::loadServlet()-> getLoader()**를 통해 클래스로더를 호출하여

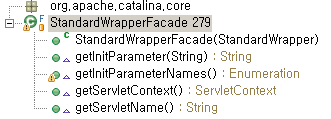
서블릿클래스를 로드한다 기자신의 로더가없으면 부모의 로더를 불러다사용한다 StandardWrapper

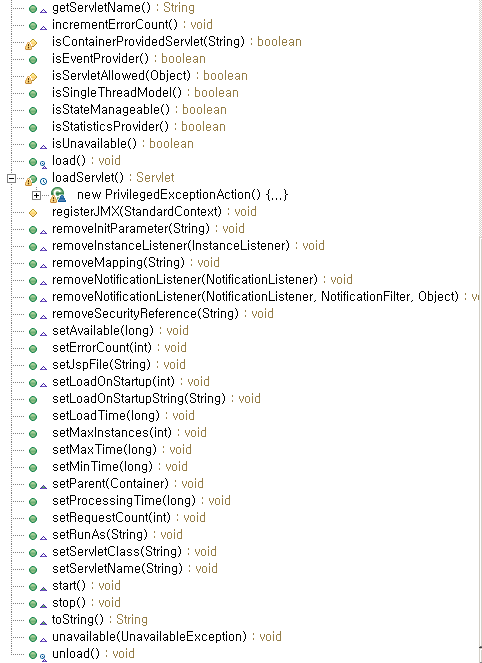






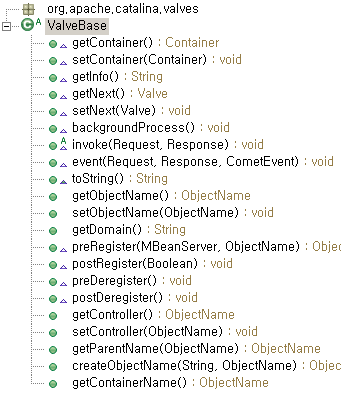
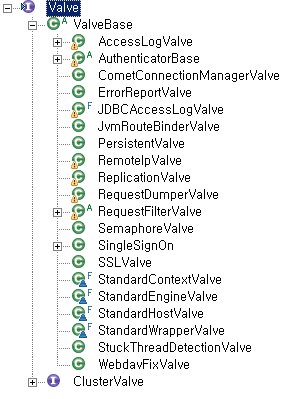
**사용자는 파사드객체를 사용한다.**

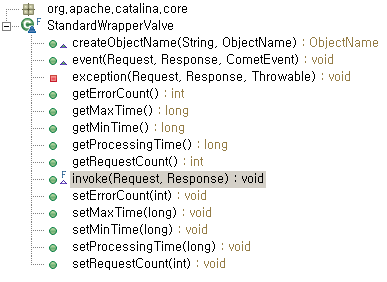




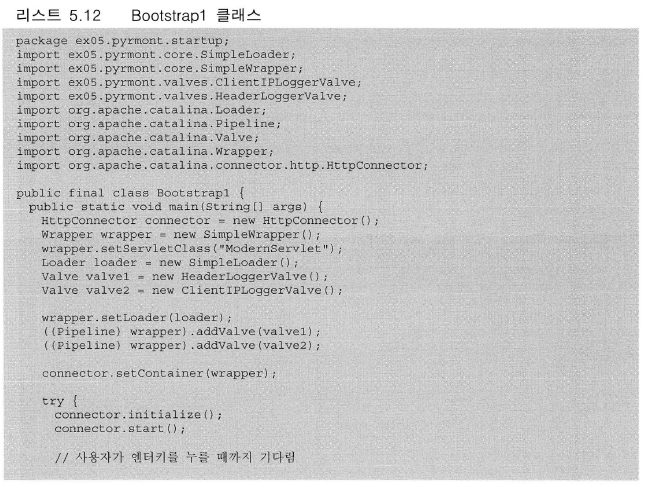


**벨브Valve**





밸브는 요청의 처리에 대한 책임이 있는 컴포넌트이며 Valve 인터페이스는 하나의 밸브를 나타낸다.



111(130)

라이프 사이클 Lifecycle

이 있다. 카탈리나의 설계 방식에서는 모든 컴포넌트들이 상위 검포넌트에 붙잡혀 있게 되기 때문에 ，부트스트랩 클래스는 오직 하나의 컴포넌트만 시작하면 된다 이러한 단일 시작/종료 메커니즘은 Lifecycle 인터페이스를 통해 가능하다.

라고 히는 배열에 저장한다.list e ners는 처음에는 아무 요소도 없는 빈 배열로 시작한다.

p r ivate Li f ecycleLi s t ener li s t eners[ ) ; new LifecycleListener[O) ;

addLifecycleListener 메소드를 통해 하나의 리스너를 추가할 때에는 기존에 있던 배 열의 요소

개수에 하나를 더한 수만큼의 크기로 새로운 배열이 생성된다 그 다음에 기존 배열의 모든 요소가 새

로운 배열에 복사되고 그에 더해 새 리스너가 추가된다. removeLifecycleListener 메 소드를 통

해 하나의 리 스너가 제거될 때에는 마찬가지로 기존에 있던 배 열의 요소 개수에서 하니를 뺀 수만큼의

크기를 갖는 새로운 배열이 생성되며 ， 제거하려는 리스너를 제외한 모든 요소들이 새 배열로 복사된다

fireLifecycleEvent 메소드는 하나의 생명주기 이벤트를 발생시킨다. 먼저 li steners 배 열

을 복제한 후， 각 요소들의 lifecycleEvent 메소드에 이벤트를 전달해 호출한다.

Lifecyc le을 구현하는 컴포넌트는 LifecycleSupport 클래스를 사용할 수 있다. 예를 들어 6장

에서 제공하는 애플리케이션에서 의 S impleContext 클래스는 다음과 같은 변수를 선언하고 있다.

prot ected Li fecycleSupport lifecycle = new Li f ecycleSupport(this) ;

SimpleContext 클래스는 생명주기 리스너 를 추가하기 위해 Lifecycl eSupport 클래 스의

addLifecycleLi stener 메 소드를 호출한다.

public void addLif ecyc l eListener(LifecycleListener listener) (

lifecycle . addLi f ecycleListener(listener ) ;

SimpleContext 클래스는 생명주기 리스너를 제거하기 위해 LifecycleSupport 클래스의

removeLifecycleListener 메 소드를 호출한다.

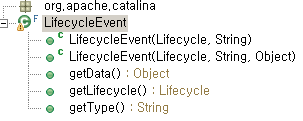
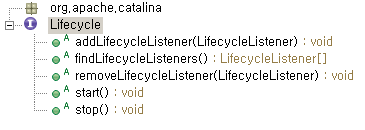
public vo i d r emoveLifecycleListener( Li fe cycleListener listener)

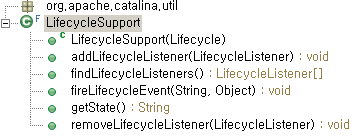
lifecycle . removeLi fecycleListener (listener) ;

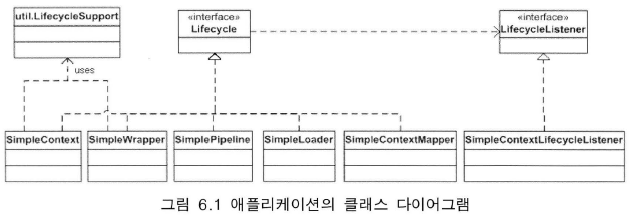
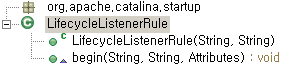
SimpleContext 클래스는 이벤트를 발생시키기 위해 다음과 같이 LifecycleSupport 클래스의

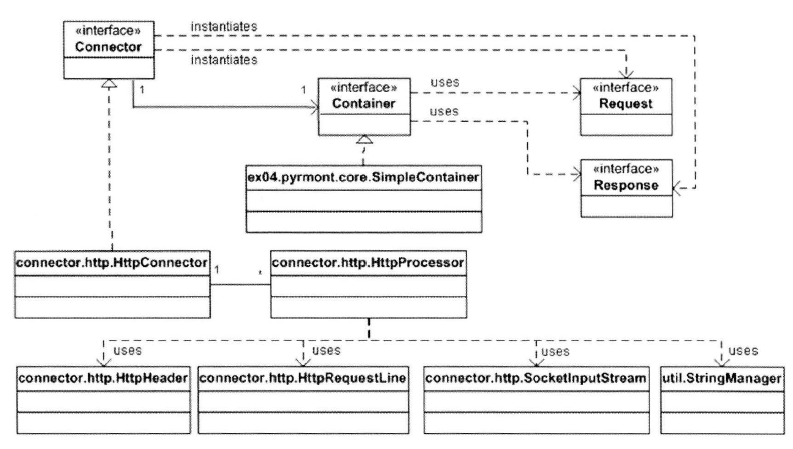
fireLifecycleEvent 메소드를 호출한다.

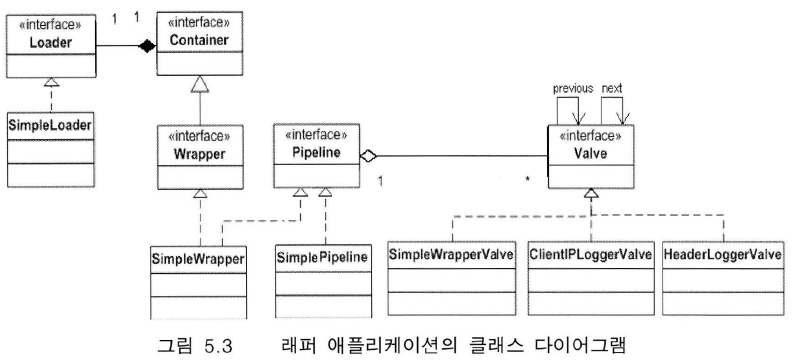
lifecyc l e.f ireLifecyc l eEvent (START\_EVENT, nul l);





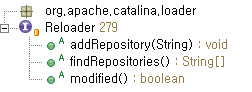






로더

톰켓이 자신만의 로더를 펼요로 하는 또 다른 이유는 WEB-INF/classes나 WEB-INF/lib 디렉토 리에 있는 클래스가 변경될 때마다 자동으로 재로딩히는 기능을 지원해야 하기 때문이다. 톰켓의 로더 를 구현히는 클래스 로더는 서블릿과 기타 클래스 따일들의 타임스탬프를 지속적으로 확인하기 위해 별도의 스레드를 사용한다. 자동 재로딩을 지원하려면 클래스 로더는 **org.apache.catalina.loader.Reloader** 인터페이스를 구현해야 한다.



**‘저장소 (repository) ’ : 클래스 로더가 찾아야 할 장소**

**‘자원 (resource) ’ : 컨텍스트의 문서 루트에 대한 정보를 갖고 있는 클래스 로더의 DirContext 객체를 말한다**

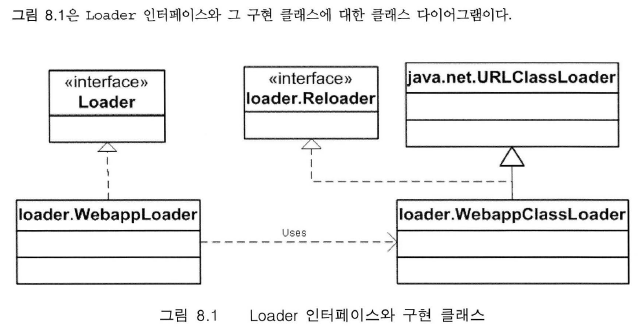
**• 클래스 로드시 특정한 규칙을 적용해야 할 때**

**• 이미 로드한 클래스를 캐싱 Ccaching) 해야 할 때**

**• 즉시 사용할 수 있게 클래스를 미리 로드해놔야 할 때**

웹 애플라케이션에서 서블릿이나 그 밖의 클래스를 로딩하는 데는 어떤 규칙이 존재한다. 예를 들어 웹 애플리케이션에서의 서블릿은 WEB- INF /classes 이하의 디렉토리에 존재하는 클래스를 사용할 수 있다. 그러나 운영중인 톰켓 J VM의 CLASSPATH에 잡혀있는 클래스라 할지라도 서블릿은 이들 클래스에 접근할 수 없어야 한다. 또 서블릿은 WEB- INF/lib 디렉토리에 있는 라이브러리 이외의 다른 라이브러리에는 접근할 수 없어야 한다

톰켓의 로더는 클래스 로더가 아니라 웹 애플리케이션 로더라고 봐야 한다. 하나의 로더는 반드시 **org.apache.catalina.Loader** 인터페이스를 구현해야 한다





WEB- INF/cla sses와 WEB-INF/lib는 저장소로서 추가돼야 하는 디렉토리이다. Loader 인

터페이스의 addRepository 메소드는 저장소를 추가할 때 사용하며， findRepositories는 모든

저장소의 배열을 리턴하는 메소드이다.

**로더는 보통 하나의 컨텍스트와 연결돼있으며**

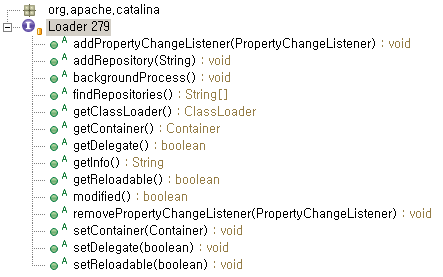
Loader 인터페이스에 정의된

getContainer와 setContainer는 이러한 연결과 관계된 메소드이다. 또 로더는 컨텍스트 내에서

하나 이상의 클래스가 변경될 경우에 재로팅하는 기능을 제공할 수 있다.

<Context pat h= " /myApp" docBase ="myApp" debug=O r eloadable= "true" / >

152(171)





**WebappLoader는 자동 재로딩 기능을 지원한다**

**public** **void** backgroundProcess()

**ContainerBase의 ContainerBackgroundProcessor**

**쓰레드 클래스를통해 지속 호출된다**

**public** **void** backgroundProcess() {

**if** (reloadable && modified()) {

**try** {

Thread.*currentThread*().setContextClassLoader

(WebappLoader.**class**.getClassLoader());

**if** (container **instanceof** StandardContext) {

((StandardContext) container).reload();

}

} **finally** {

**if** (container.getLoader() != **null**) {

Thread.*currentThread*().setContextClassLoader

(container.getLoader().getClassLoader());

}

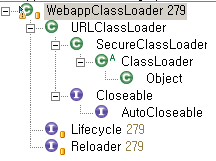
}

} **else** {

closeJARs(**false**);

}

}



WebappLoader 클래스의 start 메소드는 setRepositoryies 메소드를 호출해 클래스 로더에 저

장소를 추가한다. WEB-INF/classes 디렉토리는 클래스 로더의 addRepository 메소드로 전달되

며 ， WEB-INF/lib 디렉토리는 클래스 로더의 setJarPath 메소드로 전달된다. 이렇게 함으로써 클

래스 로더는 WEB-INF/classes 디렉토리에 있는 모든 클래스와 WEB-INF/ lib 디렉토리에 있는

모든 라이브러리를 로드할 수 있게 된다.



WebappClassLoader 클래스



캐싱

더 나은 성능을 내기 위해 이미 로드된 클래스를 캐싱함으로써， 다음 요청이 있을 때 캐시로부터 곧바

로 클래스를 사용하는 것이 가능하다. 캐성 작업은 지역적으로 이뤄지는데 ， 이 말은

WebappClassLoader 인스턴스가 캐시를 관리한다는 의미다 또한 java.lang . ClassLoader는

이전에 로드돼있던 클래스들을 Vector에 저장해 관리함으로써 이 클래스들을 가비지 컬렉션

(garbage collection) 이 되지 않게 한다. 이 경우 캐시는 상위 클래스에 의해 관리된다

**WebappClassLoader는 다음과 같은 규칙에 따라 클래스를 로드한다.**

이전에 로드된 클래스는 모두 캐시돼있으므로， 해당 클래스가 지역 캐시에 있는지를 먼저

확인한다.

• 지역 캐시에서 클래스를 찾지 못했다면 java.lang.ClassLoader의

findLoadedClass를 호출해 클래스 로더에 캐시돼있는지를 확인한다.

• 두 곳 모두에서 찾지 못했다면 웹 애플리케이션이 재정의한 J2EE 클래스를 사용하지 못

하도록 시스템 클래스 로더를 사용한다.

• SecurityManager가 사용된다면 클래스를 로드해도 되는지를 확인한다. 로드돼서는 안

되는 클래스라면 ClassNotFoundException을 발생시킨다.

• 위임 플래그가 켜 있거나(즉 delegate 변수의 값이 true 일 경우) 로드할 클래스의 패

키지가 packageTriggers 에 포함돼 있다면 클래스를 로드하기 위해 상위 클래스 로더

를 사용한다. 상위 클래스 로더가 null 이라면 시스템 클래스 로더를 사용하게 된다.

• 현재의 저장소로부터 클래스를 로드한다，

• 현재의 저장소에서 클래스를 찾을 수 없고 delegate 값이 false 라면 ， 상위 클래스

로더를 사용한다. 만약 상위 클래스 로더가 null 이라면 시스댐 클래스 로더를 사용하게된다.

• 그래도 클래스를 찾을 수 없다면 Cl assNot FoundExcept ion을 던진다

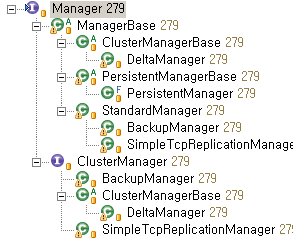
162(181)

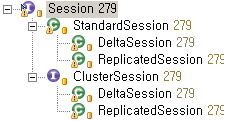
세션 관리 (매니저는 세션 객체를 관리한다. 예를 들어 매니저는 세션 객체를 생성하거나 무효화할 수 있다.)

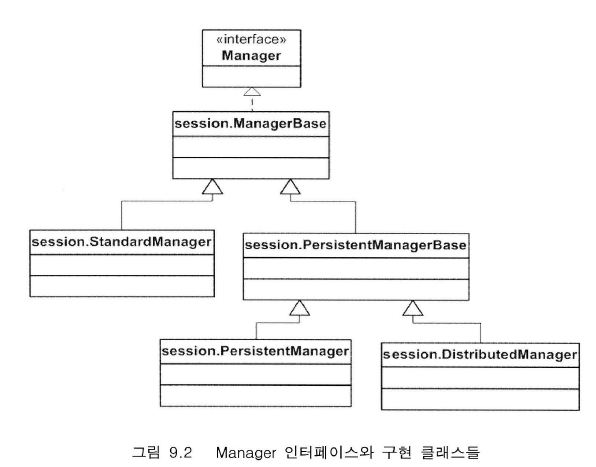
카탈리나는 매니저라는 컴포넌트를 통해 세션을 관리하는데 ， 매니저는 org.apache.catalina.Manager

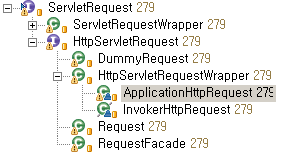
\* 1











getSession(){

... context.getManager().findSession(other.getId());

}

매니저는 기본적으로 세션 객체를 메모리에 저장한다. 그러나 톰켓은 매니저가 세션 객체를 파일이나 데이터베이스(J DBC를 통해)에 지속적으로 저장하는 것도 허용한다. 카탈리나는 세션 객체 및 세션 관리와 관련된 클래스가 포함돼있는 org.apache.catalina.session 패키지를 제공한다

ManagerBase

**protected** Map<String, Session> sessions = **new** ConcurrentHashMap<String, Session>();

**public** **void** add(Session session) {

sessions.put(session.getIdInternal(), session);

**int** size = sessions.size();

**if**( size > maxActive ) {

**synchronized**(maxActiveUpdateLock) {

**if**( size > maxActive ) {

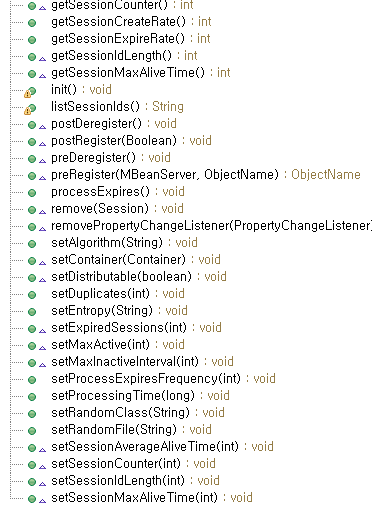
maxActive = size;

}

}

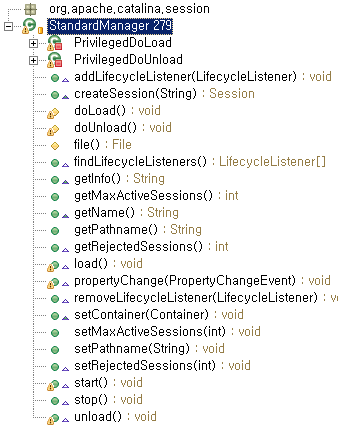
}

}



StandardManager

**protected** String pathname = "SESSIONS.ser";



Standarcl.Manager 클래스는 Manager 인터페이스의 표준 구현 클래스이며 세션 객체를 메모리에

저장하는 역할을 한다. 이 클래스는 또 Lifecycle 인터페이스 (6장 생명주기 참고) 를 구현해 시작/

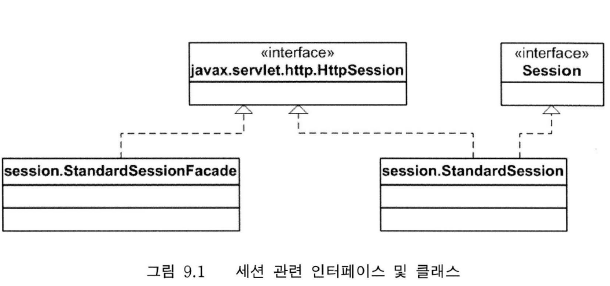
중지될 수 있다. 구현된 stop 메소드에서는 unload 메소드를 호출함으로써 유효한 Session 인스

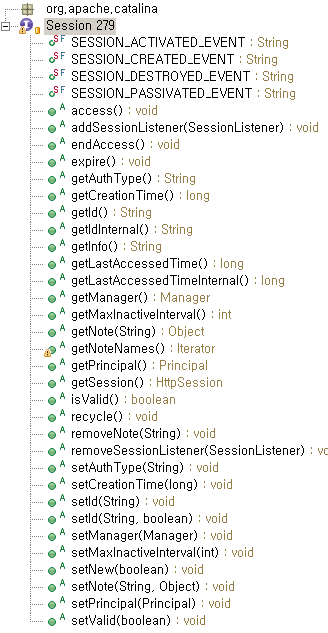
턴스들을 SESS IO NS.ser이라는 파일에 저장하게 된다.

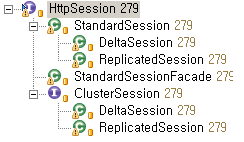
CATALINA\_HOME/work/Standalone/localhos t/examples 디 렉토리 에서 SESSIONS.ser 파일 을 찾을 수 있다.

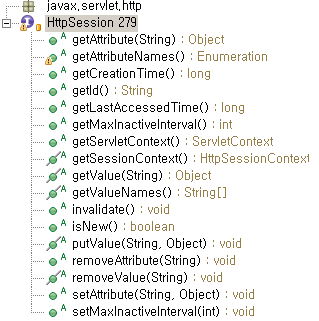
세션

서블릿 프로그래밍에 있어서 세션 객체는 javax.servlet.http.HttpSession 인터페이스로 대변된다. 이 인터페이스의 표준 구현 클래스는 org.apache.catalina.session 패커지의**StandardSession** 클래스이다 그러나 보안상의 이유 때문에 매니저는 StandardSession 인스턴스를 서블릿에 전달하지 않는다 대신 org.apache.catalina.session 패키지의StandardSessionFacade라는 퍼사드 클래스를 시용한다. 내부적으로 매니저는org.apache.catalina.Session 인터페이스와 함께 작동한다. 그림 9.1 은 세션과 관련된 클래스와 인터페이스를 나타내는 클래스 다이어그램이다.







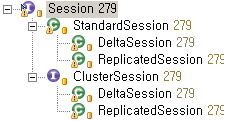


Session에서

setManager와 getManager 메소드를 사용해

Sessi on 객체를 매니저에 연결할 수 있다 또 Session 객체는 매니저와 연결돼있는 컨텍스트 내에

서 유일한 식별자를 갖는다. 세션의 식별자는 setld와 getld 메소드를 사용해 접근할 수 있다.





StandardSession, StandardSessionFacade

**public** HttpSession getSession() {

**if** (facade == **null**){

**if** (SecurityUtil.*isPackageProtectionEnabled*()){

**final** StandardSession fsession = **this**;

facade = (StandardSessionFacade)AccessController.*doPrivileged*(**new** PrivilegedAction(){

**public** Object run(){

**return** **new** StandardSessionFacade(fsession);

}

});

} **else** {

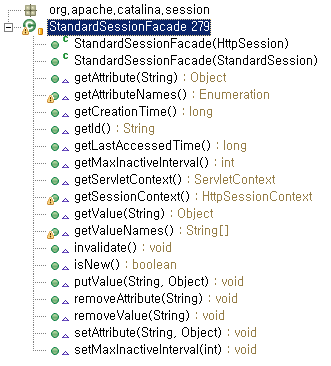
facade = **new** StandardSessionFacade(**this**);

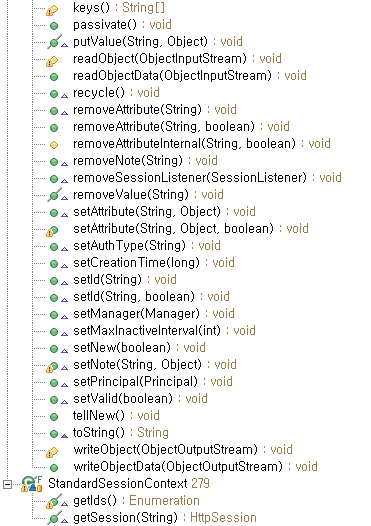
}

}

**return** (facade);

}





보안 189(208)

서블릿 컨테이너는 인증자 Cauth enticator) 라는 밸브를 통해 보안 제약 기능을 지원한다.

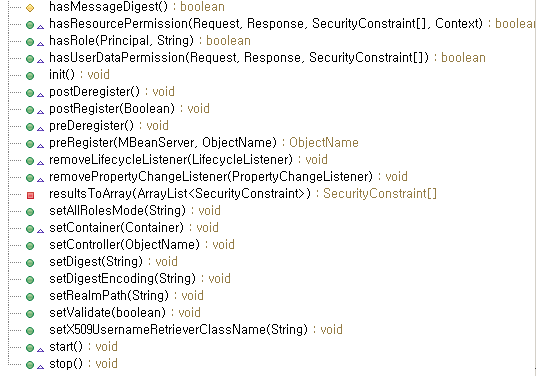
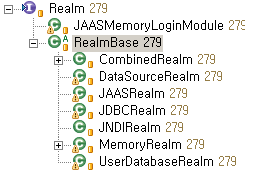
**인증자 밸브는 사용자를 인증하는 밸브이며, 래퍼 밸브보다 이전에 호출된다.** 인증에 실패하면인증자 밸브는 다음 밸브를 호출하지 않고 리턴한다.

**Realm** (영역)

‘ 영역 (realm) ’은 사용자를 인증하는 데 필요한 컴포넌트이며, 사용자 이름과 암호가 유효한지를 말해준다 1 ‘영역 ’은 **보통 하나의 컨텍스트에 연결되며**，

하나의 **컨테이너는 하나의 ‘영역 ’**만을 가질 수 있다. **컨테이너의 setRealm 메소드에 ‘ 영역’ 객체를 전달함으로써 ‘ 영역 ’을 컨테이너에 연결할 수** 있다

톰켓에서는 유효한 사용자의 정보를 기본적으로tomcat-users.xml 파일에 저장한다.



컨테이너 인터페이스 (컨테이너와 1 : 1 Realm)

**public** **void** setRealm(Realm realm);

**카탈리나의 인 증자 밸브는 사용자를 인증하기 위해 연결 돼 있는 ‘영역 ’ 객체의 authenticate 메 소드를 호출한다 (**주체는 Principal이다.)

**public** **boolean** hasRole(String role) {

**if**("\*".equals(role)) //Special 2.4 role meaning everyone

**return** **true**;

**if** (role == **null**)

**return** (**false**);

**return** (Arrays.*binarySearch*(roles, role) >= 0);

}

톰켓 5는 서블릿 2 . 4를 지원하므로 ‘모든 역할’이라는 의미의 ‘ \* , 문자를 인식할 수 있어야 한다

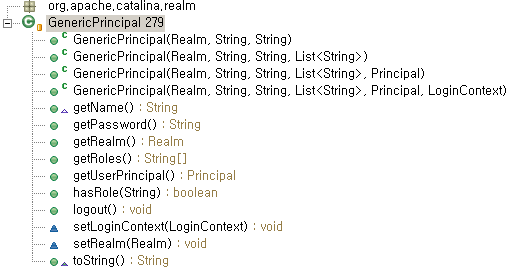
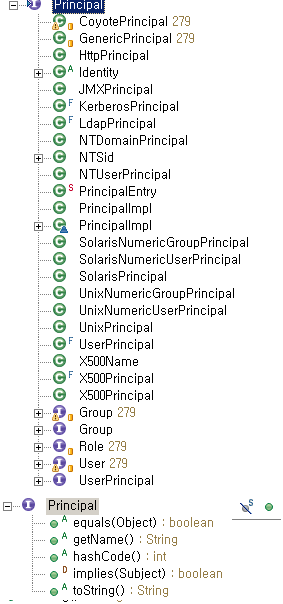
GenericPrincipal은 다음의 두 생성자 메소드에서 볼 수 있듯이 영역 과 항상 연결돼있어야 한다.

반드시 이름과 암호를 갖고 있어야 한다.

**public** GenericPrincipal(Realm realm, String name, String password) {

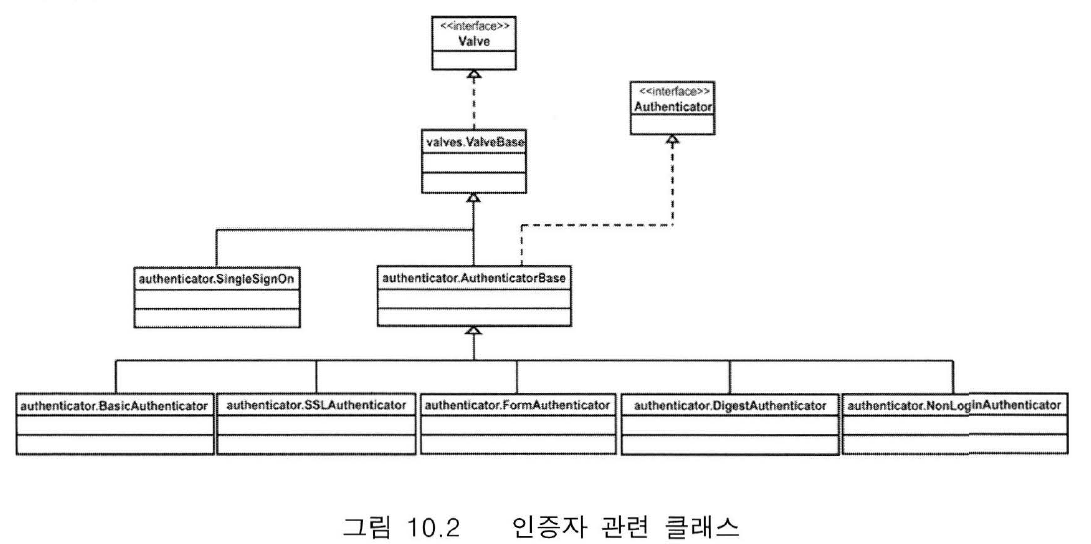
**this**(realm, name, password, **null**);

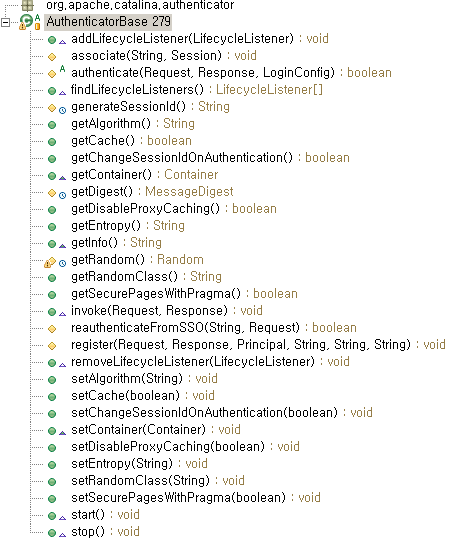
}

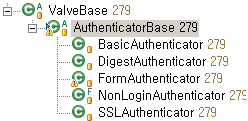


**LoginConfig** 하나의 로그인 설정은 하나의 ‘영역’ 이름을 포함하며 org.apache.ca talina .deploy. LoginConfig 클래스로 대변된다

**Authenticator** org . apache .catalina .Authent i cator 인터페이스는 인증자를 대변한다.







StandardWrapper **굉장히 중요**

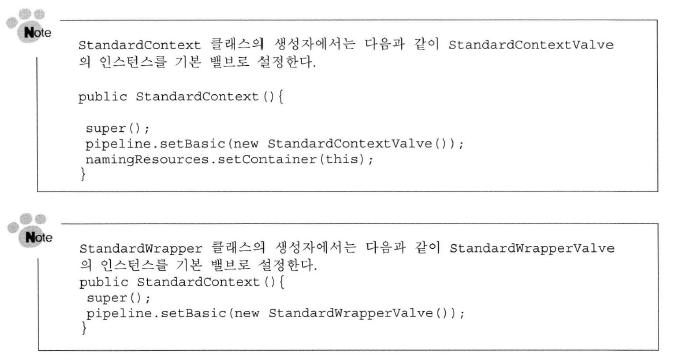
1. 커넥터는 연결돼있는 컨테이너의 invoke 메소드를 호출한다.

2. 컨태이너는 다시 모든 하위 컨테이너의 invoke 메소드를 호출한다.

3. 예를 들어 커넥터가StandardContext 인스턴스에 연결돼있다면， 커넥터는 StandardContext 의 invoke 메소드를호출할 것이며，

4. StandardContext는 다시 모든 하위 컨테이너들(이 경우 StandardWrapper 타입)의 invoke 메소드를 호출할 것이다.

(컨테이너는 하나 이상의 밸브가 포함돼있는 하나의 파이프라인을 갖는다는 것을 기억하기 바란다. )



**public** StandardContext() {

**super**();

pipeline.setBasic(**new** StandardContextValve());

broadcaster = **new** NotificationBroadcasterSupport();

}

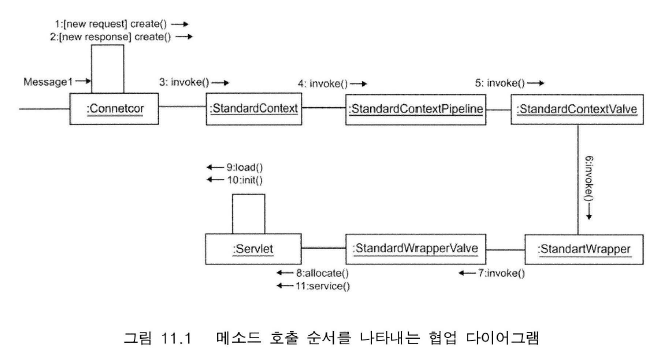
**public** StandardWrapper() {

**super**();

swValve=**new** StandardWrapperValve();

pipeline.setBasic(swValve);

..}



• 커넥터는 StandardContext 인스턴스의 invoke 메소드를 호출한다.

• StandardContext 의 i nvoke 메소드는 컨텍스트의 파이프라인의 invoke 메소드를 호출한다 StandardContext 의 파이프라인에서 기본 밸브는

Standar dContextValve 이며 ，따라서 StandardContext 의 파이프라인은 StandardContextValve 의 invoke 메소드를호출한다

• StandardContextValve 의 invoke 메소드는 요청을 처리할 적합한 래퍼를 얻은 다음，이 래퍼의 invoke 메소드를 호출한다

• StandardWrapper는 래퍼의 표준 구현 클래스이다 StandardWrapper 인스턴스의i nvoke 메소드는 자신의 파이프라인의 invoke 메소드를 호출한다，

• StandardWrapper의 파이프라인에 있는 기본 밸브는 StandardWrapperValve 이며，따라서 StandardWrapperValve 의 invoke 메소드가 호출된다.

StandardWrapperValve 의 invoke 메소드는 래퍼의 allocate 메소드를 호출해 서블릿의 인스턴스를 얻는다.

• 서블릿을 로드할 필요가 있다면 allocate 메소드는 서블릿을 로드하기 위해 load 메소드를호출한다.

• load 메소드는 서블릿의 init 메소드를 호출한다

• StandardWrapperValve는 서블릿의 service 메소드를 호출한다.

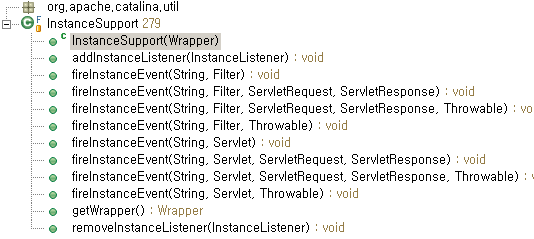
StandardWrapper

Standardwrapper 객체의 주된 역할은 자신이 대변하는 서블릿을 로드하고 인스턴스를 생성하는 것이다. 그러나 StandardWrapper가 서블릿의 service 메소드를 호출하는 것은 아니다 이 일은 **StandardWrapper의 파이프라인에 있는 기본 밸브인 StandardWrapperValve 객체가 담당한다.**

Standardwrappervalve 객체는 **StandardWrapper의 allocate 메소드를 호출해** 서블릿 인스턴스를 얻는다. 일단 서블릿 인스턴스를 받은 후에， **StandardWrapperValve는 서블릿의 service메소드를호출한다 <-???**

InstanceSupport 를통해 서블릿의 생성종료의관련된

메시지를 주고받는다.



**public** **class** StandardWrapper **extends** ContainerBase **implements** ServletConfig, Wrapper, NotificationEmitter {

**protected** StandardWrapperFacade facade = **new** StandardWrapperFacade(**this**);

**public** Servlet allocate() **throws** ServletException {

**loadServlet(){**

...

Loader loader = getLoader();

...

ClasLoader classLoader = loader.getClassLoader();

...

**if** (classLoader != **null**) {

classClass = classLoader.loadClass(actualClass);

} **else** {

classClass = Class.*forName*(actualClass);

}

...

servlet = (Servlet) classClass.newInstance();

...

servlet.init(facade); **//StandardWrapperFacade임(ServletConfig)**

...

servlet.service(req, res);

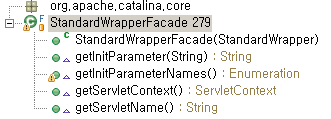
**}**

...

}



사용자는 파사드객체를 사용한다.



// Invoke jspInit on JSP pages JSP

**if** ((loadOnStartup >= 0) && (jspFile != **null**)) {

// Invoking jspInit

DummyRequest req = **new** DummyRequest();

req.setServletPath(jspFile);

req.setQueryString(Constants.***PRECOMPILE*** + "=true");

DummyResponse res = **new** DummyResponse();

**if**( Globals.***IS\_SECURITY\_ENABLED***) {

Object[] args = **new** Object[]{req, res};

SecurityUtil.*doAsPrivilege*("service",

servlet,

*classTypeUsedInService*,

args);

args = **null**;

} **else** {

servlet.service(req, res);

}

}

**final** **class** StandardWrapperValve **extends** ValveBase {

**public** **final** **void** invoke(Request request, Response response)

StandardWrapper wrapper = (StandardWrapper) getContainer();

Servlet servlet = **null**;

Context context = (Context) wrapper.getParent();

..

servlet = wrapper.allocate();

..

**if** (servlet != **null**) {

wrapper.deallocate(servlet);

}

**if** ((servlet != **null**) && (wrapper.getAvailable() == Long.***MAX\_VALUE***)){

wrapper.unload();

}

}

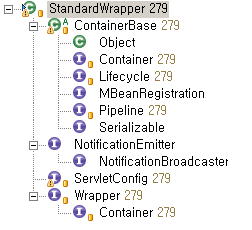
...

}

**ServletConfig 객체**

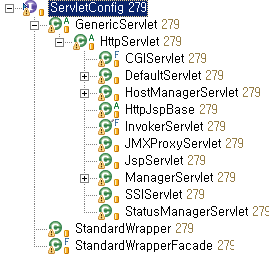
StandardWrapper 객체가 어떻게 ServletConfig 객체를 취득할 수 있는지 궁금할 것 이다.

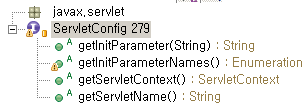
StandardWrapper 클래스. StandardWrapper 클래 스는 Wrapper 인터페이스뿐만 아니라 javax .servlet.ServletConfig 인터페이스도 구현한다.(**implements)**



**public** **class** StandardWrapper **extends** ContainerBase

**implements** ServletConfig, Wrapper, NotificationEmitter {...}





StandardWrapper

**protected** HashMap parameters = **new** HashMap(); getter,setter

<servlet>

<servlet-name>UserServlet</servlet-name>

<servlet-class>jwaf.lib.app.UserServlet</servlet-class>

<init-param>

<param-name>jwaf.config.file</param-name>

<param-value>abcdefg</param-value>

</init-param>

<init-param>

<param-name>log4j.properties</param-name>

<param-value>/WEB-INF/config/log4j.properties</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>UserServlet</servlet-name>

<url-pattern>/user.pb/\*</url-pattern>

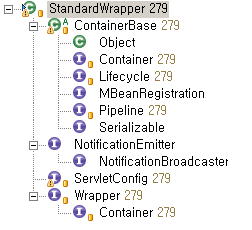
</servlet-mapping>

web.xml





StandardWrapper 안쪽내용



래퍼는 개별 서블릿에 해당하는 컨테이너이다. 따라서 래퍼는 그 하위의 컨테이너， 즉 자식 컨테이너 를 가질 수 없으며 addChild 메소드는 호출돼서는 안된다

**public** **void** addChild(Container child) {

**throw** **new** IllegalStateException

(*sm*.getString("standardWrapper.notChild"));

}

//부모추가할때도그렇다.

**public** **void** setParent(Container container) {

**if** ((container != **null**) &&

!(container **instanceof** Context))

**throw** **new** IllegalArgumentException

(*sm*.getString("standardWrapper.notContext"));

**if (container instanceof StandardContext) {**

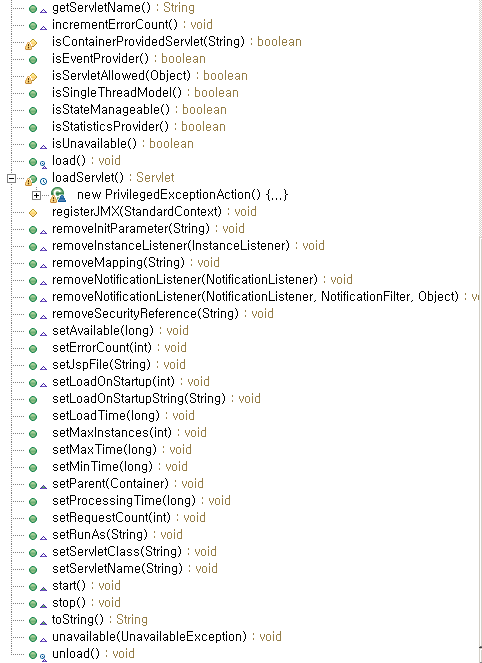
swallowOutput = ((StandardContext)container).getSwallowOutput();

unloadDelay = ((StandardContext)container).getUnloadDelay();

}

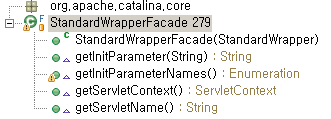
**super**.setParent(container);

}





**사용자는 파사드객체를 사용한다.**



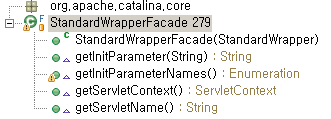
**StandardWrapperFacade**

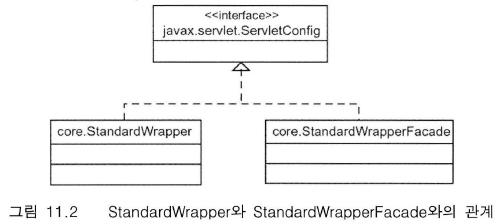
StandardWrapper 인스턴스는 자신이 로드한 서블릿의 init 메소드를 호출한다. init 메소드는 javax.servlet.ServletConfig를 필요로 하며 StandardWrapper 클래스는 그 자신이 Serv letConfig 인터 페이스를 구현하고 있으므로 이론상으로는 StandardWrapper 객체 자신을 init 메소드에 전달할 수 있다 그러나 StandardWrapper 클래스는 자신의 대부분의 메소드를 서블릿으로부터 숨길 필요가 있다 이를 위해서 StandardWrapper 클래스는 자신을 StandardWrapperFacade 인스턴스로 래핑한다.

**StandardWrapper::protected** StandardWrapperFacade facade = **new** StandardWrapperFacade(**this**);



**사용자는 파사드객체를 사용한다.**





**public** **final** **class** StandardWrapperFacade

**implements** ServletConfig {

**public** StandardWrapperFacade(StandardWrapper config) {

**super**();

**this**.config = (ServletConfig) config;

}

**private** ServletConfig config = **null**;

**private** ServletContext context = **null**;

**public** String getServletName() {

**return** config.getServletName();

}

**public** ServletContext getServletContext() {

**if** (context == **null**) {

context = config.getServletContext();

**if** ((context != **null**) && (context **instanceof** ApplicationContext))

**context = ((ApplicationContext) context).getFacade();**

}

**return** (context);

}

**public** String getInitParameter(String name) {

**return** config.getInitParameter(name);

}

**public** Enumeration getInitParameterNames() {

**return** config.getInitParameterNames();

}

}

StandardWrapperValve

StandardWrapperValve 클래스는 Standardwrapper 인스턴스의 기본 밸브이다.

**public** **class** **StandardWrapper** **extends** ContainerBase

**implements** ServletConfig, Wrapper, NotificationEmitter {

**...**

**public** StandardWrapper() {

**super**();

swValve=**new** StandardWrapperValve();

pipeline.setBasic(swValve);

...

}

...

}

**final** **class** **StandardWrapperValve** **extends** ValveBase {

**public** **final** **void** invoke(Request request, Response response) **throws** IOException, ServletException {

//StandardWrapper로부터 서블릿 인스턴스가져옴

**if** (!unavailable) {

servlet = wrapper.allocate();

}

//요청을 수신했음을 알림

**try** {

response.sendAcknowledgement();

}...

// Create the filter chain for this request 필터체인가져옴,생성

ApplicationFilterFactory factory = ApplicationFilterFactory.*getInstance*();

ApplicationFilterChain filterChain = factory.createFilterChain(request, wrapper, servlet);

..

//필터체인 호출

filterChain.doFilter(request.getRequest(), response.getResponse());

..//필터반환

filterChain.release();

..//서블릿 철회

wrapper.deallocate(servlet);

..//다시사용 안한다면

wrapper.unload();

}

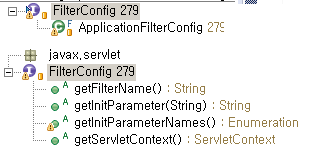
}

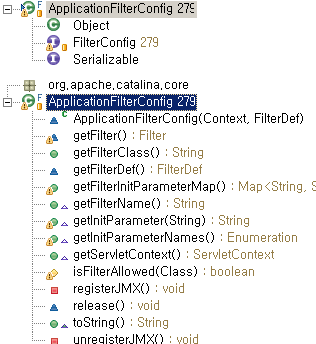
여기서 가장 중요한 것은 createFilterChain 메소드 호출과， 펼터 체인의 doFilter 메소드 호출이다. createFilterChain 메소드는 ApplicationFilterChain의 인스턴스를 생성하고， 여기에 이 StandardWrapper으로 표현되는 서블릿에 적용해야 할 모든 필터를 추가한다.

FilterDef, ApplicationFilterConfig, ApplicationFilterChain

ApplicationFilterConfig

인터페이스를 구현한다. App li cationFilterConfig는 웹 애플리케이션 시작시에 생성된 펼터 인스턴스들을 관리한다.





**if** (pos < n) {

ApplicationFilterConfig filterConfig = filters[pos++];

filter = filterConfig.getFilter();

support.fireInstanceEvent(InstanceEvent.***AFTER\_FILTER\_EVENT***,

filter, request, response);

...

}

Filter 객체를 리턴하는 getFilter

메소드를 갖고 있다. 이 메소드는 필터 클래스를 로드하고 초기화하는 역할을 한다.

// Return the existing filter instance, if any

// 기존필터있으면 넘겨라

**if** (**this**.filter != **null**)

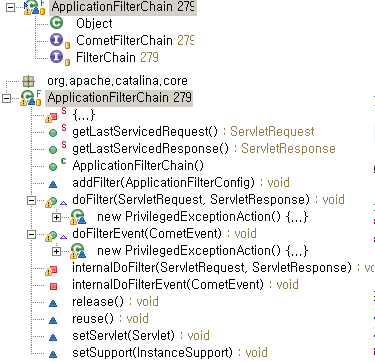
**return** (**this**.filter);

**this**.filter = (Filter) clazz.newInstance();

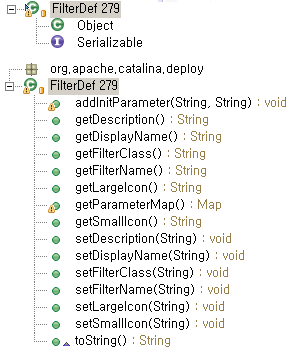
filter.init(**this**);

1 \*

ApplicationFilterChain



FilterDef 요소에 정의돼있는 펼터를 나타낸다.



<filter>

<filter-name>encodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>UTF-8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>encodingFilter</filter-name>

<url-pattern>/＊</url-pattern>

</filter-mapping>

Filter





**import** java.io.IOException;

**import** javax.servlet.Filter;

**import** javax.servlet.FilterChain;

**import** javax.servlet.FilterConfig;

**import** javax.servlet.ServletException;

**import** javax.servlet.ServletRequest;

**import** javax.servlet.ServletResponse;

**public** **class** CharsetFilter **implements** Filter {

**private** String encoding;

**public** **void** init(FilterConfig config)**throws** ServletException{

encoding = config.getInitParameter(“requestEncoding”);

**if**(encoding == **null**) encoding = “utf-8″;

}

**public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain next)**throws** IOException, ServletException{

**if**(request.getCharacterEncoding() == **null**)

{

request.setCharacterEncoding(encoding);

response.setContentType(“text/html; charset=” + encoding);

response.setCharacterEncoding(encoding);

}

next.doFilter(request, response);

}

**public** **void** destroy() {

}

}

**StandardContext**

컨텍스트는 웹 애플리케이션을 나타내며 1 개 이상의 래퍼를 포함한다 또 래퍼는서블릿에 대한 정의를 나타낸다

StandardContext 인스턴스가 생성된 후에는， 들어오는 HTTP 요청을 서비스할 수 있도록 **start 메소드가 반드시 호출돼야 한다**

start 메소드가 성공적으로 수행되기 위해서는 StandardContext 객체가 올바르게 설정돼야 한다 톰켓의 배포본에서는 수많은 StandardContext 의 설정이 이뤄진다. **StandardContext 객제는 컨텍스트가 %CATALINA\_HOME%/conf 디렉토리에 있는 기본 web.xml 내용을 파싱해** ， 배치돼있는 모든 애플리케이션에 적용할 수 있도록 준비된다또 StandardContext 인스턴스는 애플리케이션 수준의 web.xml 파일을 처리할 수 있어야 한다. 추가로 StandarContext 의 설정 과정에

**는 인증자 밸브와 증명밸브 (certificatevalve) 를 설치하는 일도 포함하고 있다**

StandardContext 클래스에는 **configured**라는 특성이 있는데 ， 이것은 StandardContext 인스턴스가 올바르게 설정됐는지의 여부를 나타내는 boolean 타입의 변수다. StandardContext는자신의 설정자 (configurator) 로서 이벤트 리스너를 사용한다. **StandardContext 의 start 메소드가 호출되면 생명주기 이벤트가 발생하게 되는데， 이 이벤트는 리스너를 호출해서StandardContext 인스턴스를 설정한다**， **설정이 올바르게 완료되면 리스너는 configured 특성을true로 바꾼다**. 설정이 성공하지 못하면 StandardContext 인스턴스는 구동되는 것을 거부할 것이며 ， 따라서 HTTP 요청을 서비스하지 못하게 될 것이다.

StandardContext 인스턴스는 들어오는 모든 HTTP 요청에 대해 서비스할

준벼가 됐다는 의미다. available의 값이 false 인 경우는 이와 반대의 의미다.

StandardContext 클래스는 boolean 타입의 configured라고 하는 변수를 갖는데， 이 변수 의 초기값은 false 이다.

**생명주기 리스너가 StandardContext 인스턴스를 설정하는 일을**

**성공적으로 마치면 StandardContext 의 configured의 값을 true로**

지정할 것이다.

StandardContext는 성공적으로 구동하게 되며 ， conf igured의 값이 false라면 stop 메소드를 호출함으로써 start 메소드가 수행되는 동안 시작됐던 모든 컴포넌트들을 중지시킨다

**public** **class** StandardContext **extends** ContainerBase **implements** Context, Serializable, NotificationEmitter{

**public** StandardContext() {

**super**();

pipeline.setBasic(**new** **StandardContextValve**());

broadcaster = **new** NotificationBroadcasterSupport();

}

**public** **synchronized** **void** start() **throws** LifecycleException {

...

**if** (ok) {

**if** (*log*.isDebugEnabled())

*log*.debug("Starting completed");

setAvailable(**true**);

} **else** {

*log*.error(*sm*.getString("standardContext.startFailed", getName()));

**try** {

stop();

} **catch** (Throwable t) {

*log*.error(*sm*.getString("standardContext.startCleanup"), t);

}

setAvailable(**false**);

}...

}

}

StandardContext::start

**public** **class** StandardContext **extends** ContainerBase **implements** Context, Serializable, NotificationEmitter{

**..**

**public** **synchronized** **void** start() **throws** LifecycleException {

• BEFORE START EVENT 이벤트 발생

• available 특성의 값을 false로 지정

• configured 특성의 값을 false로 지정

• 자원준비

• 로더 준비

• 매니저 준비

• 문자셋 맵퍼 초기화

• 이 컨텍스트와 연관된 다른 컴포넌트 구동

• 하위 컨테이너 (래퍼) 구동

• 파이프라인 구동

• 매니저 구동

• START EVENT 이벤트 발생 . 여기서 리스너 (ContextConfig) 는 몇 가지 설정 작업을 수

행한다 (15장에서 설명하겠다) . 설정작업이 성공적으로 완료되면 ContextConfig는

StandardContext 인스턴스의 configured 값을 true로 바꾼다

• configured 값 확인. 그 값이 true라면 p o stWelcomePages 메소드를 호출하고， 구동시에 로드할 하위

래퍼들을 로드하며 ， available 특성을 true로 지정한다. configured값이 false 라면 stop 메소드를 호출한다.

• AFTER START EVENT 이벤트 발생

}

재로딩 지원

StandardContext 클래스는 애플리케이션의 재로딩이 허용됐는지를 나타내는 reloadable 특성

을 정의하고 있다. 재로딩이 허용된 애플리케이션이라면 web.xml 파일이 변경되거나 WEB-INF/classes 디렉토리 내에 하나 이상의 클래스가 재검파일될 경우 재로딩이 일어날 것이다.

Sta ndardContex t 클래 스는 애플리케이션을 재로드하는 데 있어서 자신의 로더에 의존한다. 톰

켓 4에서는 StandardContext 의 로더는 WebappLoader이다. Loader의 구현 클래 스인

WebappLoader는 WEB- INF 디렉토리의 모든 클래스들과 JAR 파일들의 타임 스탬프를 확인하는

스레드를 갖고 있다. 이 스레드를 실행할 때 알아야 할 것은 setCont a i ner 메 소드를 호출해

WebappLoade r를 StandardCont ext 에 연결하는 것뿐이다. 다음은 톰켓 4 에서 Web a ppLo ade r

클래스의 setContainer 메 소드이다.

자세히 안나와있어서 우선 패스

호스트와 엔진

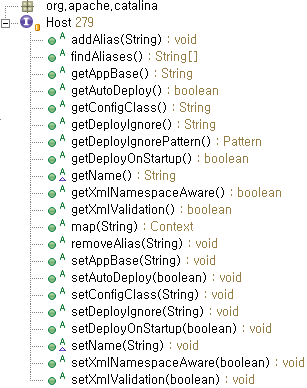


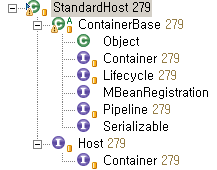
엔진은 카탈리나 서블릿 엔진 전체를 나타낸다. 엔진이 사용되면 **엔진은 항상 최상위의 컨테이너**가 된

다 엔진에 추가되는 하위 컨테이너는 보통 org.apache.catalina.Host나

org . apache . catalina.Context를 구현한 컨테이너이다， 기본적으로 톰켓 배포본에서는 엔진이

사용되며 이 **엔진은 하나의 기본 호스트를 갖는다.**





StrandardHost 스탠다드호스트는 start할때 ErrorReportValvClass를 addValve하게된다.

들어오는 모든 요청마다 호스트의 invoke 메소드가 호출되는데， StandardHost는 자체적으로 invoke 메소드를 구현하고 있지 않기 때문에 그 상위 클래스인

ContainerBase 의 invoke 메소드가 호출될 것이 다

이 invoke 메소드는 다시 StandardHost 의 기본 밸브인

StandardHostValve 의 invoke 메소드를 호출한다.

**public** **abstract** **class** ContainerBase...{

...

**public** **void** invoke(Request request, Response response)**...**{

pipeline.getFirst().invoke(request, response);

}

...

**public** **synchronized** **void** addValve(Valve valve) {

pipeline.addValve(valve);

fireContainerEvent(***ADD\_VALVE\_EVENT***, valve);

}

...

}

**public** **class** StandardHost **extends** ContainerBase **implements** Host{

**private** String errorReportValveClass = "org.apache.catalina.valves.ErrorReportValve";

**public** StandardHost() {

**super**();

**pipeline.setBasic(new StandardHostValve());**

}

...

**public** **synchronized** **void** start() **throws** LifecycleException {

...

init();

**if**( mserver.isRegistered(realmName ) ) {

mserver.invoke(realmName, "init",

**new** Object[] {},

**new** String[] {}

);

}

}

...

**if** ((errorReportValveClass != **null**) && (!errorReportValveClass.equals(""))) {

**try** {

**boolean** found = **false**;

**if**(errorReportValveObjectName != **null**) {

ObjectName[] names =

((StandardPipeline)pipeline).getValveObjectNames();

**for** (**int** i=0; !found && i<names.length; i++)

**if**(errorReportValveObjectName.equals(names[i]))

found = **true** ;

}

**if**(!found) {

Valve valve = (Valve) Class.*forName*(**errorReportValveClass**)

.newInstance();

**addValve(valve);**

errorReportValveObjectName = ((ValveBase)valve).getObjectName() ;

}

} **catch** (Throwable t) {

*log*.error(*sm*.getString

("standardHost.invalidErrorReportValveClass",

errorReportValveClass), t);

}

}

...

**super**.start();

}

}

**StandardHost에서 올바른 Context를 찾기위해 map메서드를 사용한다**

**public** Context map(String uri) {

...

Context context = **null**;

String mapuri = uri;

**while** (**true**) {

context = (Context) findChild(mapuri);

**if** (context != **null**)

**break**;

**int** slash = mapuri.lastIndexOf('/');

**if** (slash < 0)

**break**;

mapuri = mapuri.substring(0, slash);

}

// If no Context matches, select the default Context

**if** (context == **null**) {

context = (Context) findChild("");

}

**if** (context == **null**) {// Complain if no Context has been selected

*log*.error(*sm*.getString("standardHost.mappingError", uri));

**return** (**null**);

}

**if** (*log*.isDebugEnabled())// Return the mapped Context (if any)

*log*.debug(" Mapped to context '" + context.getPath() + "'");

**return** (context);

}

**ContextConfig**는 애플리케이션의 web. xml 파일의 위치를 알아야 한다. ContextConfig는 applicationConfig 메소드의 web.x ml 파일 읽기를 시도한다. 다음은 applicationConfig 메소드의 일부다

class Constants ...{...

**public** **static** **final** String ***Package*** = "org.apache.catalina.startup";

**public** **static** **final** String ***ApplicationContextXml*** = "META-INF/context.xml";

**public** **static** **final** String ***ApplicationWebXml*** = "/WEB-INF/web.xml";

**public** **static** **final** String ***DefaultContextXml*** = "conf/context.xml";

**public** **static** **final** String ***DefaultWebXml*** = "conf/web.xml";

**public** **static** **final** String ***HostContextXml*** = "context.xml.default";

**public** **static** **final** String ***HostWebXml*** = "web.xml.default";

...}

Constants.***DefaultContextXml***;

Constants.***DefaultWebXml***;

Constants.***ApplicationWebXml***

Constants.***HostWebXml***

Constants.***HostContextXml***

**protected** **synchronized** **void** start() {

defaultWebConfig();

applicationWebConfig();

**if** (!context.getIgnoreAnnotations()) {

applicationAnnotationsConfig();

}

**if** (ok) {

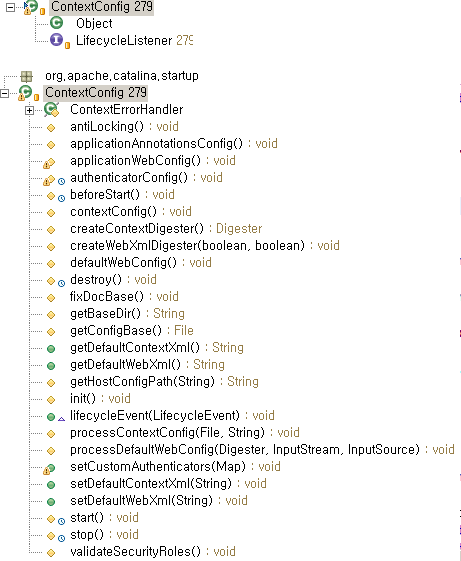
validateSecurityRoles();

}

**if** (ok)

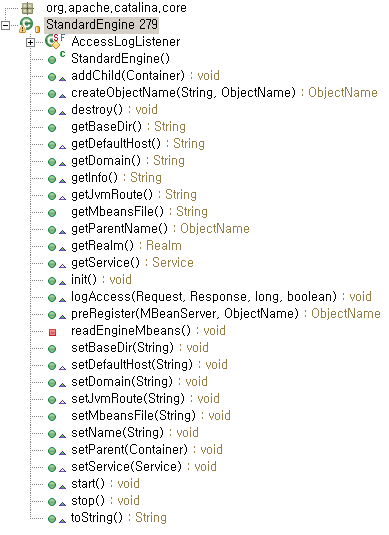
authenticatorConfig();

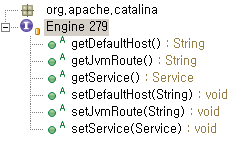
}

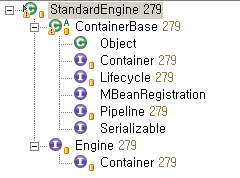


StandardEngine

org . apache . catalina .core.StandardEngi ne은 Engine 인터페이 스의 표준 구현 클래 스이다 StandardEngine은 StandardContext 및 StandardHo st 에 비해 상대 적으로 작은 클래스 이다‘ 다음 생 성자 메소드에서 볼 수 있듯이 StnadardEngine 인스턴스가 생성될 때 기본 밸브가추가된다 StandardEngineValve







StandardEngine, StandardEngineValve

**final** **class** StandardEngineValve

**extends** ValveBase {...

**public** **final** **void** invoke(Request request, Response response)

**throws** IOException, ServletException {

// Select the Host to be used for this Request

Host host = request.getHost();

**if** (host == **null**) {

response.sendError

(HttpServletResponse.***SC\_BAD\_REQUEST***,

***sm***.getString("standardEngine.noHost",

request.getServerName()));

**return**;

}

// Ask this Host to process this request

host.getPipeline().getFirst().invoke(request, response);

}

...

}

invoke 메소드는 요청 및 응답 객체에 대한 검증을 끝낸 후 요청을 처리하는 데 사용할 Host 인스턴스를 얻는다. Host는 Engine 의 map 메소드를 통해 취득하며 ， Host를 얻으면 Hos t 의 invoke메소드를 호출하게 된다.

**public** **class** StandardEngine **extends** ContainerBase **implements** Engine {

**public** StandardEngine() {

**super**();

**pipeline.setBasic(new StandardEngineValve());**

/\* Set the jmvRoute using the system property jvmRoute \*/

**try** {

setJvmRoute(System.*getProperty*("jvmRoute"));

} **catch**(Exception ex) {

}

// By default, the engine will hold the reloading thread

backgroundProcessorDelay = 10;

}

...

}

서버 (서버와 서비스)

**org.apache.catalina.Server 인터페이스는 카탈리나 서블릿 컨테이너 전체를 나타내며**， **다른모든 컴포넌트를 포함한다**. ‘서버’가 특별히 유용한 점은 전체 시스템을 시작하고 종료할 수 있는 세련된 메커니즘을 제공한다는 사실이다. 시스랩을 종료시키기를 원한다면 정해진 포트에 종료 명령을 전송하면 된다. 서버가 올바른 종료 명령을 받아들였다면 시스템의 모든 컴포넌트를 종료시킬 것이다. 시작 및 종료 메커니즘의 원리는 이렇다. 우리가 ‘서버 ’를 구동시키면， ‘서버’는 자신에 포함되어 있는 모든 컴포넌트들을 구통시킨 후， 종료 명령을 무한정 기다리는 것이다. 만약 시스댐을 종료시키기를 원한다면I 정해진 포트에 종료 명령을 보내면된다





StandardServer

**public** **void** initialize() **throws** LifecycleException{

초기화...

**for** (**int** i = 0; i < services.length; i++) {

services[i].initialize();

}

...

}

**public** **void** start() **throws** LifecycleException {

...

**synchronized** (services) {

**for** (**int** i = 0; i < services.length; i++) {

**if** (services[i] **instanceof** Lifecycle)

((Lifecycle) services[i]).start();

}

}...

}

**public** **void** await() {

클라이언트(고객아님)커넥션을 기다리고(:8085)

데이터가 숏다운이면숏다운시킴

}

**public** **void** addService(Service service) {

service.setServer(**this**);

Service results[] = **new** Service[services.length + 1];

System.*arraycopy*(services, 0, results, 0, services.length);

results[services.length] = service;

services = results;

...

**if** (initialized) {

service.initialize();

}

...

((Lifecycle) service).start();

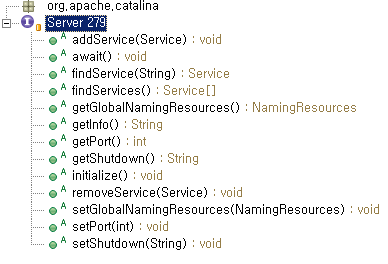
support.firePropertyChange("service", **null**, service);

}

}

서버는 서비스를 갖지 않거나

1개 이상의 서비스를 가질 수 있다.



서비스

StandardService

**public** **void** start() **throws** LifecycleException {

...

// Start our defined Container first

**if** (container != **null**) {

**synchronized** (container) {

**if** (container **instanceof** Lifecycle) {

((Lifecycle) container).start();

}

}

}

**synchronized** (executors) {

**for** ( **int** i=0; i<executors.size(); i++ ) {

executors.get(i).start();

}

}

**synchronized** (connectors) {

**for** (**int** i = 0; i < connectors.length; i++) {

**try** {

((Lifecycle) connectors[i]).start();

} **catch** (Exception e) {

*log*.error(***sm***.getString(

"standardService.connector.startFailed",

connectors[i]), e);

}

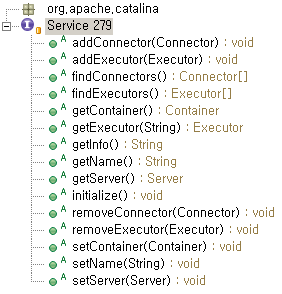
}

}...

}

StandardService 의 start 메소드는 **컨테이너와 모든 커넥터를 시작하게 한다**，







컨테이너와 커넥터

StandardService 클래스는 Container 인스턴스에 대한 객체 레퍼런스인 container 변수와，

모든 커넥터들의 배열에 해당하는 connectors 라는 변수를 사용한다., **setContainer메서드에 컨테이너와 커넥터를 연결시킨다.**

Service service = new StandardService()

service.setName ( "Stand- a l one Service");

Server server = new StandardServer() ;

server. addService(service) ;

service . addConnector(connector);

service.setContainer (engine) ;

server.initialize();

server.start();

server.awiat()

**public** **class** StandardService **implements** Lifecycle,Service, MBeanRegistration{

...

**protected** Connector connectors[] = **new** Connector[0];

**protected** Container container = **null**;

...

**public** **void** setContainer(Container container) {

Container oldContainer = **this**.container;

**if** ((oldContainer != **null**) && (oldContainer **instanceof** Engine))

((Engine) oldContainer).setService(**null**);

**this**.container = container;

**if** ((**this**.container != **null**) && (**this**.container **instanceof** Engine))

((Engine) **this**.container).setService(**this**);

**if** (started && (**this**.container != **null**) &&

(**this**.container **instanceof** Lifecycle)) {

**try** {

((Lifecycle) **this**.container).start();

} **catch** (LifecycleException e) {

;

}

}

**synchronized** (connectors) {

**for (int i = 0; i < connectors.length; i++)**

**connectors[i].setContainer(this.container);**

}

**if** (started && (oldContainer != **null**) &&

(oldContainer **instanceof** Lifecycle)) {

**try** {

((Lifecycle) oldContainer).stop();

} **catch** (LifecycleException e) {

;

}

}// Report this property change to interested listeners

support.firePropertyChange("container", oldContainer, **this**.container);

}

**public** **void** addConnector(Connector connector) {...

connector.initialize();

((Lifecycle) connector).start(); ...

}

}

1 :

\*

컨테이너

여기선 엔진이들어간다.

서비스

커넥터

1 : \*

1 : \*

서버

이 서비스와 연결된 컨테이너 는 각 커넥터의 setContainer 메소드에 전달돼 ，

이 서비 스에 속해 있는 컨테이너와 각 커넥터가 연결 되게 한다.

다이제스터

XML->오브젝트화 시켜주는 아파치오픈프로젝트

**객체의 생성**

Digester 인스턴스가 어떤 패턴에 대응하는 객체를 생성하기를 원한다면 a ddObjectCreate 메소

드를 사용하면 된다 이 메소드에는 4개의 중복 정의된 메소드가 있다， 그 중 가장 많이 사용되는 2

<?xml version= "1.0" encoding="ISO-8859-1"?>

<employee firstName= "Brian" lastName="May">

<office>

<address streeName="Wellington Street" streetNumber="110"1>

</office>

</employee>

개의 메소드의 시그너쳐는 다음과 같다.

**public void addObjectCreate(java . lang . String pattern , java.lang.Class c lazz )**

**public void addObjectCreate (java . lang.String pattern , java.lang ‘ String className)**

이들 메소드에 패턴과 Class 객체나 클래스 이름을 전달해 호출한다. 예를 들어 Digester가

employee 라는 패턴을 만나서 Employee 라는 객체 (ex15.pyrmont.digestertest.Employee)

를 생성하게 하고 싶다면 다음과 같은 코드를 작성하면 된다.

**digester .addObjectCreate( "e mployee " , ex15 . pyrmont.digestertest.Employee.class)**

또 다음과 같은 코드를 작성해도 된다.

**digester.addObjectCreate("employee" , "ex15 . pyrmont . digestertest . Employee " );**

나머지 2개의 addObjectCreate 메소드는 클래스의 이름을 메소드의 인자로 넘기는 대신 . XML 요

소에 클래스를 정의할 수 있게 한다 이것은 대단히 강력한 특정이라고 할 수 있는데 ， 클래스의 이름

이 런타임에 결정되기 때문이다 다음은 이 두 메소드의 시그너쳐이다

**public void addObjectCreate(java.lang.String pattern ,**

**Java ‘ lang.String cla ssName , java.lang.String attributeName)**

**publ ic void addObjectCreate (java. lang.String pattern ,**

**java.lang.String attributeName , java . lang.Class clazz)**

이 두 메소드에서 attributeName 이라는 인자는 인스턴스를 생성할 클래스의 이름을 갖고 있는

XML 요소의 속성 이름을 말한다. 예를 들면 다음과 같이 사용할 수 있다

**digester . addObjectCreate( “ employee" , null , “ className") ;**

여기서 속성의 이름은 className 이 된다.

다음에는 XML 요소에 다음과 같이 클래스 이름을 지정할 수 있다.

**<employee firstName="Brian" lastName= "May"**

**className="ex15.pyrmont . digestertest . Employee ">**

또 다음과 같이 addObjectCreate 메소드에 기본값으로 사용할 클래스 이름을 전달할 수도 있다. \*따로 다이제스트 알아보도록합니다.

**ContextConfig** 인스턴스는 기본 web.xml 파일과 애플리케이션

web.xml 파일을 읽고 파싱해 XML 요소를 자바 객체로 변환시켜준다는 것이다. 기본 web.xml 파

일은 **CATALINA\_HOME 아래의 conf 디렉토리에 위치한다.** 이 파일은 기본 서블릿

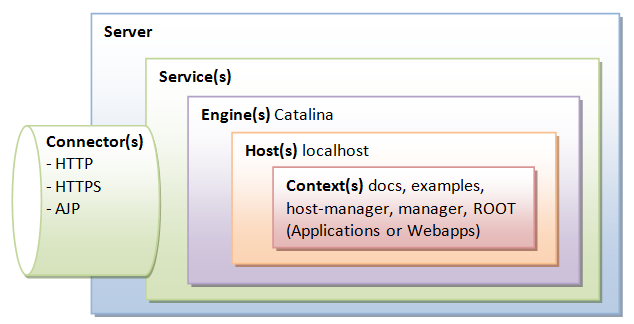
(DefaultServlet) 의 정의와 뱀핑， 파일 확장자와 MIME 타입의 뱀핑， 기본 세션 만료시간 설정 ，

환영 파일 목록 등에 관해 기술하고 있다 l 기본 web.xml 파일을 열어서 내용을 직접 확인하기 바란다.

애플라케이션 web.xml 파일은 애플리케이션에 대한 설정 파일이며， 해당 애플리케이션의 WEBINF

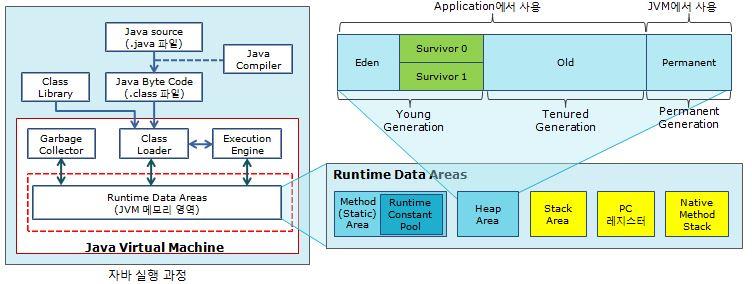
디렉토리에 존재한다. 기본 web.xml과 애플리케이션 web.xml은 필수로 존재해야 하는 파일이

아니다. 이 두 파일이 존재하지 않는다 하더라도 ContextConfig는 진행을 계속할 것이다.



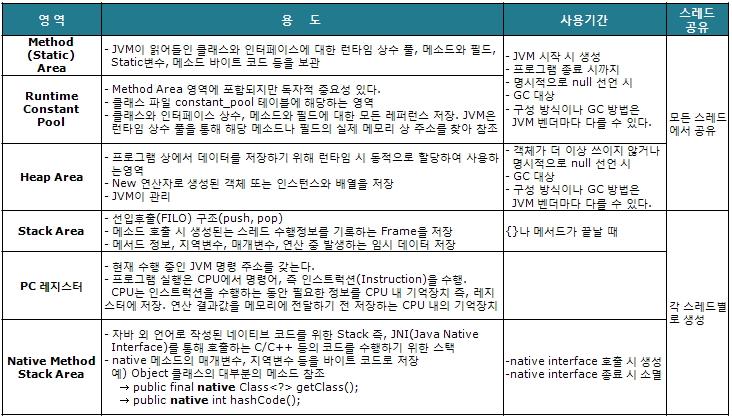
## JVM과 메모리 구조

### 1. JVM 구조



- **Class loader** : JVM내로 class를 로드하고 Link를 통해 적절히 배치하는 일련의 작업을 수행하는 모듈. Runtime 시 동적으로 class load   
- **Execution Engine** :  class loader를 통해 JVM 내의 런타임 데이터 영역에 배치된 바이트 코드는 실행 엔진에 의해 실행. 실행 엔진은 자바 바이트 코드를 명령어 단위로 읽어서 실행   
- **Garbage Collector** : JVM은 Garbage Collector를 통해 자동화된 메모리 관리 기능을 제공. Garbage Collector는 Garbage Collection을 수행하는 모듈 쓰레드를 말한다. Application이 생성한 객체의 생존 여부를 판단해서 더 이상 사용되지 않는 객체를 해제함으로써 메모리 관리가 자동화되록 한다.   
- **Runtime Data Areas** : JVM 이라는 프로그램이 운영체제 위에서 실행되면서 할당받는 메모리 영역. 런타임 데이터 영역은 6개의 영역으로 나뉜다. 

### 2. Runtime Data Areas



#### ※ Stack Area - method call Tarce 추적 샘플

아래 샘플 소스는 JVM Stack Area 영역에 저장되는 호출 메소드의 스택을 차례대로 출력한 내용이다. 정확히 말하자면 Statck Area에는 메소드가 수행되는 정보 Frame(순서대로 호출된 method 참조값, 변수 값등등이 담긴 프레임) 정보가 있다. 아래의 샘플은 현재 호출된 메소드가 위치한 스레드의 문맥(Context)에서 메소드만 호출 순서별로 출력한 내용이다. (자세한 소스는 파일로 첨부한 소스를 참조) 

[?](http://www.gliderwiki.org/wiki/76)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79 | public class Subcontracting {      . . . . .      OrderSupplier gaab;      . . . . .      public void startWork() {          gaab.work();      }        public class OrderSupplier {      . . . . .          Contractor    ul;          . . . . .          public void changeBizRoll() {              . . . . .          }          . . . . .          public void order() {              ul.work();          }            public void work() {              order();              . . . . .              changeBizRoll();          }      }        public class Contractor implements ProjectWork {          . . . . .          SecondContractor byung;          . . . . .          public void work(boolean bCommonSense) {              . . . . .              byung.work(bCommonSense);          }          . . . . .      }        public class SecondContractor implements ProjectWork {          . . . . .          Workers       developer;          . . . . .          public void work(boolean bCommonSense) {              //. . . . .              developer.work(bCommonSense);          }            public void work() {              work(true);          }      }        public class Workers implements ProjectWork  {          . . . . .          public void overTimeWork() {   // 야근 작업 work stack trace 추적              . . . . .              StackTraceElement[] ste = new Throwable().getStackTrace();              for(StackTraceElement element : ste) {                     System.out.println(element.getClassName() + " ["+ element.getMethodName() +"]" +                              "(line:"+ element.getLineNumber() + ")");              }          }            public void work(boolean bCommonSense) {              . . . . .              if (bCommonSense) {   // 정상 작업 work stack trace 추적                  . . . . .                  StackTraceElement[] ste = new Throwable().getStackTrace();                  for(StackTraceElement element : ste) {                      System.out.println(element.getClassName() + " ["+ element.getMethodName() +"]" +                              "(line:"+ element.getLineNumber() + ")");                  }              } else {                  overTimeWork();              }          }          . . . . .      }  } |

StackTrace 추척 실행 결과  
*/\* - 최종 호출 method에서 최초 호출 method 순으로 표시됨 \*/*/\* - 정상 작업 work stack trace 추적 : 최종 호출 method에서 최초 호출 method 순으로 \*/   
mentalcollapse.basic.jvm.Subcontracting$Workers [work](line:197)   
mentalcollapse.basic.jvm.Subcontracting$SecondContractor [work](line:151)   
mentalcollapse.basic.jvm.Subcontracting$Contractor [work](line:108)   
mentalcollapse.basic.jvm.Subcontracting$Contractor [work](line:112)   
mentalcollapse.basic.jvm.Subcontracting$OrderSupplier [order](line:66)   
mentalcollapse.basic.jvm.Subcontracting$OrderSupplier [work](line:70)   
mentalcollapse.basic.jvm.Subcontracting [startWork](line:27)   
mentalcollapse.basic.jvm.StackTraceTest [main](line:8)   
*/\* - 야근 작업 work stack trace 추적 \*/   
mentalcollapse.basic.jvm.Subcontracting$Workers [overTimeWork](line:184)   
mentalcollapse.basic.jvm.Subcontracting$Workers [work](line:204)   
mentalcollapse.basic.jvm.Subcontracting$SecondContractor [work](line:151)   
mentalcollapse.basic.jvm.Subcontracting$Contractor [work](line:108)   
mentalcollapse.basic.jvm.Subcontracting$OrderSupplier [changeBizRoll](line:61)   
mentalcollapse.basic.jvm.Subcontracting$OrderSupplier [work](line:72)   
mentalcollapse.basic.jvm.Subcontracting [startWork](line:27)   
mentalcollapse.basic.jvm.StackTraceTest [main](line:8)*

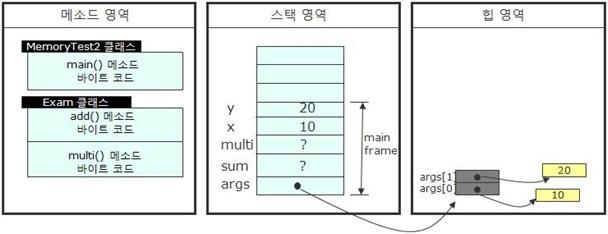
### *3. 객체 생성과 메모리*

*-실행예제*

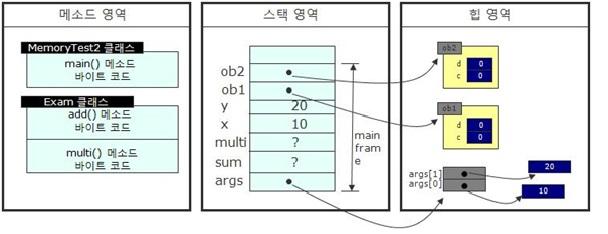
[*?*](http://www.gliderwiki.org/wiki/76)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | public class Exam {      int c, d;      public int add(int a, int b) {          c = a + b;          return c;      }      public int multi(int a, int b) {          d = a \* b;          return d;      }  }    public class MemoryTest2 {      public static void main(String[] args) {          int sum, multi;          int x, y;          x = Integer.parseInt(args[0]);          y = Integer.parseInt(args[1]);  // 1번          Exam obj1 = new Exam();          Exam obj2 = new Exam();         // 2번          sum = obj1.add(x, y);          multi = obj2.multi(x, y);          System.out.println("add = " + sum);          System.out.println("multipy = " + multi);      }  } |

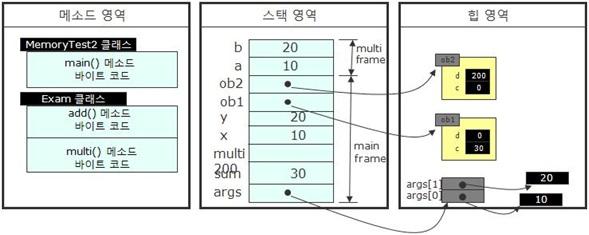
*1) main() 메소드가 수행되는 시점   
- “1번” 위치   
- 프로그램 실행 시 두 개의 매개변수 지정(10,20)   
- 두 개의 매개변수를 힙 영역에 문자열 객체로 저장   
- x,y 에는 정수로 변환된 값이 저장*

**

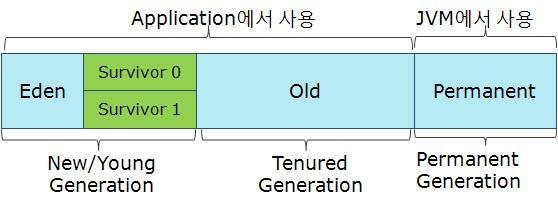
*2) 객체가 생성되는 시점   
- “2번” 위치   
- ob1, ob2 객체가 힙 영역에 생성   
- 각각의 객체에는 두 개의 객체 속성 변수가 저장*

**

*3) 객체의 메소드 수행 시점   
- 두 개의 객체가 생성된 다음 add() 메소드와 multi() 메소드가 수행된다   
- 아래 그림은 add() 메소드 수행 후 multi() 메소드가 수행되는 시점*

**

### *4. Heap Area*

**

*-****Young Generation****: 이 영역은 java 객체가 생성되자마자 저장되고, 생긴지 얼마 안되는 객체가 저장되는 공간. Java 객체가 생성되면 이 영역에서 저장되다가, 시간이 지남에 따라 우선순위가 낮아지면 Old 영역으로 옮겨진다. 이 영역에서 객체가 사라질 때 Minor GC가 발생한다.   
  
-****Old (Tenured) Generation****:  Young Generation 영역에서 저장되었던 객체 중에 오래된 객체가 이동되어서 저장되는 영역. 이 영역에서 객체가 사라질 때 Major GC(Full GC)가 발생한다.   
  
-****Permanent 영역****: Class loader에 의해 load되는 Class, Method 등에 대한 Meta 정보가 저장되는 영역으로 JVM에 의해서 사용된다.   
Reflection을 사용하여 동적으로 클래스가 로딩되는 경우에 사용된다. 내부적으로 Reflection 기능을 자주 사용하는 Spring Framework를 이용할 경우 이 영역에 대한 고려가 필요하다.*

*참고 출처 : http:*stophyun.tistory.com/37

[**synchronized, volatile, native, transient - Java**](http://lkhstory.blogspot.kr/2008/01/synchronized-volatile-native-transient.html)

**synchronized**   
하나의 method 혹은 method의 일부만 적용될 수 있는 제한자로 해당 method 혹은 부블을 한번에 한 thread만 접근 할 수 있도록 하는 기능을 제공합니다. thread-safe method를 만들기 위해 사용하는 제한지입니다.  
  
**volatile**  
변수에만 사용할 수 있는 제한자이며, 변수의 값이 비동기적으로 변경될 수 있는 값임을 알립니다.  
  
**native**   
method에만 사용할 수 있는 제한자로 이 method가 자바가 아닌 다른 native 언어로 구현된 메소드임을 나타냅니다.  
  
**transient**   
변수에만 사용할 수 있는 제한자로 이 제한자가 붙어 있는 변수는 직렬화(serialization) 대상에서 제외됩니다.

### [**native/ transient/ synchronized/ volatile 제한자와 위치**](http://iptvspot.blogspot.kr/2010/03/native-transient-synchronized-volatile.html)

출처 : <http://blog.naver.com/lcs4you?Redirect=Log&logNo=130024200464>

**[출처]** [제 2장 6.7. native/ transient/ synchronized/ volatile 제한자와 위치](http://blog.naver.com/lcs4you/130024200464)|**작성자** [멋진놈](http://blog.naver.com/lcs4you)

이절에서 소개하는 제한자는 자주 사용되지는 않는다. 하지만 이 제한자들이 어떤 용도로 사용되는지에 대해서는 알으시면 될것 같습니다.    이 제한자들 또한 자바 키워드임을 명심하자.

native 제한자

 - native 제한자(modifier)는 자바가 아닌 다른 언어로 작성된 코드를 자바에서 사용하기 위한 것이다.

    이제한자는 반드시 메소드에만 선언되어야 한다.

transient 제한자

 - transient 제한자는 객체가 직렬화되는 과정에서 해당 필드가 저장되지 않아야 한다는 것을 알리기

    위해 사용된다. 이 제한자는 반드시 멤버 변수에만 선언되어야 한다.

synchronized 제한자

 - synchronized 제한자는 코드의 같은 블록을 하나 이상의 스레드가 동시에 접근하는 것을 막기 위

    해 사용된다. 이 제한자는 반드시 메소드나 블록에 선언되어야 한다.

volatile 제한자

 - volatile 제한자는 변수가 스레드에 의해 비동기적으로 변경될 수 있음을 알리기 위해 사용된다. 이

    제한자는 final 변수를 제외한 변수에 선언될 수 있다.

제한자들은 선언될 수 있는 위치가 정해져 있기 때문에 선언 기능 위치에 대해서 정확히 알고 선언하는 것이 중요하다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 제한자 | 변수 | 메소드 | 클래스 | 내부 클래스 |
| public | ○ | ○ | ○ | ○ |
| private | ○ | ○ | × | ○ |
| protected | ○ | ○ | × | ○ |
| final | ○ | ○ | ○ | ○ |
| abstract | × | ○ | ○ | ○ |
| static | ○ | ○ | × | ○ |
| native | × | ○ | × | × |
| transient | ○ | × | × | × |
| volatile | ○ | × | × | × |
| synchronized | × | ○ | × | × |

※ 제한자의 선언 위치

**[출처]** [제 2장 6.7. native/ transient/ synchronized/ volatile 제한자와 위치](http://blog.naver.com/lcs4you/130024200464)|**작성자** [멋진놈](http://blog.naver.com/lcs4you)

출처 : <http://shoutrock.egloos.com/4102389>

**transient**필드 앞에 사용할 수 있는 지정 예약어로 해당 필드가 입시로 사용됨을 나타낸다.  
transient가 사용된 필드의 경우 네트워크 전송이나 파일에 저장시에 메모리 자체가 없는 것과 같은 효과를 나타낸다.  
  
transient는 직렬화시킬 수 없는 멤버 필드나 전송하기 싫은 멤버 필드를 위해서 사용한다.  
실제 클래스에 중요한 데이터 멤버에 transient를 선언하는것은 정말 뷁이다.  
  
  
**volatile**  
  
멤버 필드로만 사용할 수 있고, 지역 변수로는 사용할 수 없는 것으로 일반적으로 메모리 변수라고 한다.  
실제 메모리에 할당되어 프로그램 흐름에 관계없이 인터럽트 루틴이나 CPU에 의해 값이 변환될 수 있는 필드로 사용된다.  
final과 함께 사용되기도 한다

출처 : <http://blog.naver.com/byung2love?Redirect=Log&logNo=140020203781>

\* transient

- 객체를 저장할때 (ex:파일) 이 키워드를 사용한 변수는 저장되지 않음.

\* volatile

- 공유자원(변수)의 경우 동시접근으로 값이 업데이트 되는 경우가 발생하는데

이 키워드를 사용한 변수는 업데이트시 락(Lock)을 걸어버림.

(synchronized키워드를 사용한 것과 동일)

- 주의: 자바 1.5 이전버전에서는 동기화가 잘 되지않음...

**[출처]** [transient, volatile키워드](http://blog.naver.com/byung2love/140020203781)|**작성자** [하얀방울](http://blog.naver.com/byung2love)

출처 : <http://lkhstory.blogspot.com/2008/01/synchronized-volatile-native-transient.html>

**synchronized**  
하나의 method 혹은 method의 일부만 적용될 수 있는 제한자로 해당 method 혹은 부블을 한번에 한 thread만 접근 할 수 있도록 하는 기능을 제공합니다. thread-safe method를 만들기 위해 사용하는 제한지입니다.  
  
**volatile**  
변수에만 사용할 수 있는 제한자이며, 변수의 값이 비동기적으로 변경될 수 있는 값임을 알립니다.  
  
**native**  
method에만 사용할 수 있는 제한자로 이 method가 자바가 아닌 다른 native 언어로 구현된 메소드임을 나타냅니다.  
  
**transient**  
변수에만 사용할 수 있는 제한자로 이 제한자가 붙어 있는 변수는 직렬화(serialization) 대상에서 제외됩니다.

출처 : <http://blog.naver.com/iioi20?Redirect=Log&logNo=50727881>

**native 수정자** : 메서드 호출이 아닌 운영체제 종속적인 메서드를 호출한다.

**synchronized 수정자** : 쓰레드 동기화와 관련된 지시어. 한번에 하나의 쓰레드에서만 접근이 가능.

**transient 수정자** : 객체 직렬화시에 데이터의 보존여부를 결정하는 수정자

**volatile 수정자** : 값이 수시로 변경 가능하다는 것을 지시. 컴파일러가 변수 최적화를 하지 않게 해줌.

**[출처]** [native 수정자, synchronized 수정자, transient 수정자, volatile 수정자](http://blog.naver.com/iioi20/50727881)|**작성자** [iioi20](http://blog.naver.com/iioi20)

자바클래스로더

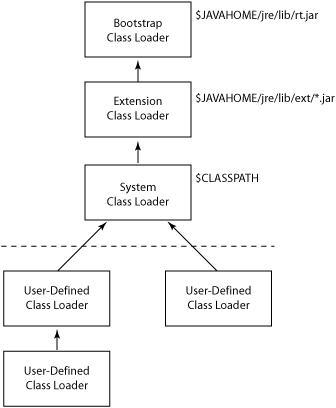
클래스 로더는 자바의 핵심 라이브러리와 CLASSPATH 환경

변수에 잡혀있는 모든 라이브러리를 찾는다. 그러나 펼요한 클래스를 찾지 못했을 경우엔

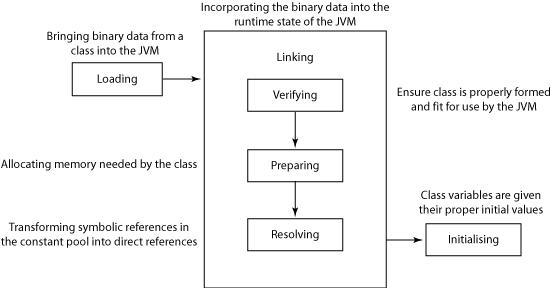
java.lang.ClassNotFoundException을 발생시키게 된다.

**부트스트랩 클래스로더**  
부트스트랩 클래스로더는 전문적으로 말해서 클래스로더가 아니다. 왜냐면 부트스트랩 클래스로더는 JVM의 네이티브 코드 영역에 존재하며, Object와 같은 코어 자바 클래스를 VM에 로딩할 때 사용되기 때문이다. 부트스트랩 클래스로더는 sun.boot.class.path 프로퍼티에 지정되어 있는 값을 이용하여 자바 런타임 라이브러리를 찾는다. 이 값을 명시적으로 지정하지 않을 경우, [자바 2 디렉토리]/jre/lib/rt.jar 파일로부터 자바 런타임 클래스들을 로딩한다.  
  
JDK 1.0이나 JDK 1.1.x 때부터 착실하게(?) 자바를 공부해왔던 개발자라면 누구나 [JDK디렉토리]/lib/classes.zip 파일을 CLASSPATH 환경변수에 추가해주었을 것이다. 하지만, 자바2에서 JDK 1.0이나 JDK 1.1.x 때와는 달리 CLASSPATH 환경변수나 명령행의 옵션인 -classpath에 자바 런타임 클래스들을 추가해줄 필요가 없다. 왜냐면, 부트스트랩 클래스로더가 자동적으로 읽어오기 때문이다.  
  
**sun.misc.Launcher$ExtClassLoader**  
ExtClassLoader는 익스텐션 클래스로더(extension classloader)라고도 불리며, 자바의 확장 클래스들을 로딩할 때 사용된다. ExtClassLoader는 URLClassLoader 클래스를 상속하며, java.ext.dirs 프로퍼티에서 지정한 디렉토리에 위치한 .jar 파일로부터 클래스를 읽어온다. 이 프로퍼티의 값을 명시적으로 지정하지 않으면, 기본적으로 [자바 2 디렉토리]/jre/lib/ext 디렉토리에 위치한 .jar 파일로부터 클래스를 읽어온다.  
  
**sun.misc.Launcher$AppClassLoader**  
AppClassLoader는 시스템 또는 어플리케이션 클래스로더라고 부르며, java.class.path 프로퍼티에 명시된 경로에서 코드를 로딩하는 클래스로더이다. ExtClassLoader과 마찬가지로 URLClassLoader를 상속하고 있다. CLASSPATH에 있는 각각의 디렉토리나 .jar 파일은 URL로 변환되어 AppClassLoader에 전달되며, AppClassLoader의 생성자에서는 이 URL들을 상위 클래스인 URLClassLoader 생성자에 전달한다.  
  
ClassLoader.getSystemClassLoader() 메소드를 호출할 때, 이 클래스로더가 리턴된다. 개발자가 작성한 대부분의 클래스들은 이 클래스로더를 통해서 로딩된다. 또한, AppClassLoader는 ExtClassLoader를 부로 클래스로더 지정하고 있기 때문에, 어플리케이션에서 기본적으로(즉, AppClassLoader를 통해서) 익스텐션 디렉토리에 있는 Jar 파일로부터 클래스들을 읽어올 수 있다.  
  
**sun.applet.AppletClassLoader**  
이름에서도 알 수 있듯이, AppletClassLoader는 웹 브라우저가 웹 페이지에서 사용되는 애플릿의 바이트 코드를 다운로드 한 후, 그 애플릿을 실행하는 것을 목적으로 하는 클래스로더이다. AppletClassLoader는 URL을 사용하여 HTTP, FTP 또는 파일 시스템으로부터 클래스를 로딩하기 때문에, URLClassLoader를 상속하고 있다. 하지만, 많이 사용하고 있는 웹 브라우저인 IE나 Netscape의 경우, AppletClassLoader가 아닌 그 웹 브라우저만의 애플릿 클래스로더를 구현하고 있기 때문에, 브라우저마다 서로 다른 동작을 보일수도 있다.  
  
**java.security.SecureClassLoader**  
SecureClassLoader 클래스의 주요 목적은 JVM에 바이드코드를 로딩하고 사용하는 것에 대한 보안을 제어하는 것이다. 하지만, 이 클래스는 실제로 클래스 코드를 로딩할 수 있는 안전한 방법을 제공하지 않으며, 다른 클래스로더가 확장할 수 있는 베이스 클래스로서의 역할을 한다. 따라서 이 클래스는 자바 런타임 라이브러이에 있는 많은 클래스로더의 상위 클래스이며, 대표적인 것으로 URLClassLoader를 들 수 있다. 참고적으로, 이 클래스를 추상 클래스이기 때문에, 직접적으로 이 클래스의 인스턴스를 생성해서 사용할 수 없다.  
  
**java.rmi.server.RMIClassLoader**  
RMIClassLoader는 ClassLoader가 아니며, RMI 런타임 시스템에서 클래스의 로딩과 마샬링(marshaling)을 처리해주는 래퍼 클래스(warpper class)이다. 실제로, RMIClassLoader는 sun.rmi.serer.LoaderHandler 클래스와의 간단한 브릿지(bridge)이다. 실제 클래스의 로딩은 LoaderHandler 클래스의 이너(inner) 클래스로 존재하는 로더 클래스들을 통해서 이루어진다. 이 이너 로더 클래스는 URLClassLoader를 상속하고 있다. 실제로 엔터프라이즈 환경에서는 RMI와 관련된 부분에서만 이 클래스로더가 자동적으로 사용될 뿐, 이 클래스로더를 직접적으로 사용하는 경우는 거의 없다. 왜냐면, URLClassLoader 자체가 HTTP, FTP와 같은 URL을 통해서 클래스를 로딩할 수 있도록 해 주기 때문이다.

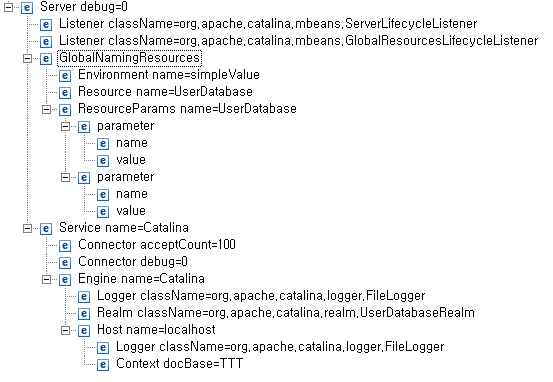
3. Class Loader 위임  
   -



옆의 그림은 class loader delegation model 이다.      이는 Loading요청을 서로 에게      전달 하는 ClassLoader의 그래프     이다.   
  
     옆의 그림과 같이 Class Loader     들은 하나의 delegation parent와     함께생성되고 다음 위치 에서     Class를 찾는다.   
  
     \* 다음 위치란  
**Cache, Parent, Self**  
  
    CalssLoader가 최초 하는 일은    전에도 같은 Class를 Loading    하도록 요청을 받았는지를    먼저 파악한다.   
  
    만약 요청을 받은적이 있으면,     Cache에 저장된 Class를     리턴 한다.     그렇지 않을 경우,     Parent Class를 Loading 할    기회를 준다.   
  
    이러한 두 단계들이 반복되고 Depth First 방식으로 탐색이 진행 된다.   
    만약 Parent Class에도 해당 Class가 존재하지 않으면 (ClassNotFoundException)   
    ClassLoader는 Class의 소스에 대한 공유(작업한 코드)의 경로 (Self)를 검색한다.   
  
    Parent ClassLoader는 Class를 처음으로 loading할 기회가 주어지기 때문에 Root에   
    가장 가까운 ClassLoader에 의해 로딩 된다.   
  
   가장 핵심적인 bootstrap class들은 java.lang.Object같은 정확한 클래스 버전이 로딩되었는지를   
   확인하고, Class Loader가 그 자체 또는 부모(Parent)에 의해서 Loading 되었는지를   
   Class가 불 수 있도록 한다. 위에서 말했다 싶이 자식에 의해 Loading된 Class 들은 볼 수 없다.   
  
  
   - Bootstrap Class Loader  
   다른 ClassLoader들과는 달리 **Bootstrap Class Loader**는 자바 코드에 의해서  
   instance화 될 수 없다.   
   Bootstrap ClassLoader 는 Boot ClassPath에서 핵심 시스템 Class들을 로딩하는데  
   이것은 일반적으로 JavaHome의 jre/lib 디렉토리에 있는 JAR 파일들이다.   
   단 **-Xbootclasspath**옵션으로 classpath를 추가 수정할 수 있다.   
  
  
   - Extension Class Loader  
   주로 jre/lib/ext 디렉토리에 위치한 Extension 디렉토리에서 Class 들을 Loading 한다.   
   사용자의 classpath를 수정하지 않고 다양한 보안 확장 같은 새로운 확자으로 쉽게   
   갈 수 있는 기능을 제공 한다.   
  
   - System Class Loader   
   CLASSPATH 환경 변수에 지정된 결로에서 코드를 로딩 한다.   
   기본적으로 이 Class Loader는 사용자가 생성한 모든 ClassLoader의 Parent 이다.   
  
  
     
4. Class Loding 단계   
   - Class Loadig은 Loading, Linking, initialzing 단계로 나누어 진다.   
  
   - 대부분의 Class Loading과 관련된 문제는 이 세단계 중 한 단계에서 발생한다.   
  
   - 로딩 단계는 필요한 Class File을 배치하고 ByteCode로 Loading하는 단계 이다.   
  
   - JVM의 로딩 프로세스는 Class 객체에 매우 기본적인 메모리 구조를 제공한다.  
     메소드, 필드, 기타 참조 Class 들이 이 단계에서는 다루어지지 않는다. 



   - Linking은 세 단계 중에서 가장 복잡한 단계이다. 다름과 같이 세 개의 주요 단계들로 나뉜다.   
     ㄱ. ByteCode 확인 - ByteCode가 정상적으로 작동 하는지 확인 한다.   
     ㄴ. Class Preparation - Class 에서 정의된 필드, 메소드, 인터페잇 들을 나타내는   
                                       데이터 구조를 준비한다.   
     ㄷ. Resolving - Class 에 다양한 방식으로 참조된 모든 Class들을 Loading한다.   
                         - super class, interface, fild, method signature, local fild  
  
   - initialize 단계 동안, Class 내에 포함된 정적 initializer들이 실행된다. 이 단계의 끝에 가서는   
     기본 값으로 초기화 된다.   
  
   - 이 세 단계를 지나면 Class는 완전히 로딩되고 사용할 준비가 된다.   
     Class Loading은 Lazy방식으로 수행 될 수 있기 때문에 일부 Class Loading Processor 는   
     Loading 시점외에 처음 클래스를 사용할 때 수행될 수도 있다.      
  
  
5. Explicit Loading vs Implicit Loading  
   - 클래스가 로딩되는 두가지 방식의 차이   
   - Explicit : 다음과 같은 메소드 호출들 중 하나를 사용하여 로딩될 때 발생한다.   
                  - (java.lang.ClassLoader).loadClass()  
                  - Class.forName()  
   - 이러한 메소드들 중 하나가 호출 되면 이름이 인자로 지정된 클래스가 클래스 로더에 의해서   
     로딩된다.   
     그렇지 않을 경우, 로더는 클래스를 로딩하기 위해 delegation model을 사용한다.   
   
   - Implicit : Class 가 레퍼런스, 인스턴스화, 명시적 메소드 호출을 제외한 상속의 경우 로딩되는 방식  
                   이 경우 슬그머니 로딩작업은 초기화되고 JVM은 필요한 레퍼런스를 결정하여   
                   만약 로딩이 되어 있으면 레퍼런스만 리턴되고 그렇지 않을 경우 델리게이션 모델을   
                   사용하여 클래스를 로딩하게 된다.

server.xml

|  |
| --- |
| <Server debug=*"0"* port=*"8005"* shutdown=*"SHUTDOWN"*>  <Listener className=*"org.apache.catalina.mbeans.ServerLifecycleListener"* debug=*"0"*/>  <Listener className=*"org.apache.catalina.mbeans.GlobalResourcesLifecycleListener"* debug=*"0"*/>  <GlobalNamingResources>  <Environment name=*"simpleValue"* type=*"java.lang.Integer"* value=*"30"*/>  <Resource auth=*"Container"* description=*"User database that can be updated and saved"* name=*"UserDatabase"* type=*"org.apache.catalina.UserDatabase"*>  </Resource>  <ResourceParams name=*"UserDatabase"*>  <parameter>  <name>factory</name>  <value>org.apache.catalina.users.MemoryUserDatabaseFactory</value>  </parameter>  <parameter>  <name>pathname</name>  <value>conf/tomcat-users.xml</value>  </parameter>  </ResourceParams>  </GlobalNamingResources>    <Service name=*"Catalina"*>  <Connector acceptCount=*"100"* connectionTimeout=*"20000"* debug=*"0"* disableUploadTimeout=*"true"* enableLookups=*"false"* maxSpareThreads=*"75"* maxThreads=*"150"* minSpareThreads=*"25"* port=*"80"* redirectPort=*"8443"*/>  <Connector debug=*"0"* enableLookups=*"false"* port=*"8009"* protocol=*"AJP/1.3"* redirectPort=*"8443"*/>  <Engine debug=*"0"* defaultHost=*"localhost"* name=*"Catalina"*>  <Logger className=*"org.apache.catalina.logger.FileLogger"* prefix=*"catalina\_log."* suffix=*".txt"* timestamp=*"true"*/>  <Realm className=*"org.apache.catalina.realm.UserDatabaseRealm"* debug=*"0"* resourceName=*"UserDatabase"*/>  <Host appBase=*"webapps"* autoDeploy=*"true"* debug=*"0"* name=*"localhost"* unpackWARs=*"true"* xmlNamespaceAware=*"false"* xmlValidation=*"false"*>  <Logger className=*"org.apache.catalina.logger.FileLogger"* directory=*"logs"* prefix=*"localhost\_log."* suffix=*".txt"* timestamp=*"true"*/>  <Context docBase=*"TTT"* path=*"/TTT"* reloadable=*"true"* source=*"org.eclipse.jst.j2ee.server:TTT"*/></Host>  </Engine>  </Service>  </Server> |